Entwicklung eingebetteter Software mit UML

Der Do-it-Prozess V 1.0

ANTJE VON KNETHEN
JÜRGEN MÜNCH

Entwicklung eingebetteter Software mit UML: Der Do-it-Prozess V 1.0 /
von Antje von Knethen und Jürgen Münch, 1999.
Interner Bericht Nr. 05/00, Sonderforschungsbereich 501.

{vknethen,muench}@informatik.uni-kl.de

Die Autoren möchten sich bei Prof. H. Dieter Rombach für die wissenschaftliche Unterstützung sowie die Bereitstellung des Umfeldes, in dem die vorliegende Arbeit entstanden ist, herzlich bedanken. Der weitere Dank gilt den Kollegen der AG Software Engineering sowie des Sonderforschungsbereichs 501 an der Universität Kaiserslautern. Dies sind insbesondere Raimund L. Feldmann und Stefan Vorwieger, die bei der Entwicklung des Beispielsystems als Betreuer mitgewirkt haben, Thomas Schneider, der die Systemadministration unterstützt hat, sowie Lothar Baum, Martin Becker, Lars Geyer, Martin Kronenburg und Georg Molter, die beratend zur Seite standen. Der Dank schließt Wolfgang Wagenbichler ein, der das Dokumentenmanagement in diesem Projekt übernommen und hierfür unterstützende Werkzeuge entwickelt hat, sowie die Studenten der Software-Engineering-Praktika, die das Beispielsystem entwickelt haben. Darüber hinaus gilt Dr. Barbara Paech und Christian Bunse vom Fraunhofer Institut für Experimentelles Software Engineering Dank für wertvolle Kommentare zur Prozessbeschreibung.

Sonderforschungsbereich 501
Arbeitsgruppe Software Engineering
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-67653 Kaiserslautern

© 2000 Antje von Knethen und Jürgen Münch, Kaiserslautern.

Alle Rechte vorbehalten.

Inhalt

KAPITEL 1 *Einführung* ***1***

1.1 Zielsetzung **2**
1.2 Charakterisierung des Entwicklungsprozesses **4**
1.3 Herkunft des Prozesses **8**
1.4 Prozessbeschreibung **8**
1.5 Der Prozess im Überblick **13**
1.6 Das Anwendungsbeispiel **15**
1.7 Planung **15**
1.8 Weiterführende Literatur **18**

KAPITEL 2 *Projekt initialisieren* ***21***

2.1 Prozessverfeinerung **22**
2.2 Produkte **24**
Problembeschreibung ***25***
B-Anforderungen ***26***
Domänenwissen ***28***

2.3 Rollen **30**
2.4 Kontrollfluss **31**
2.5 Prozess “Problembeschreibung bearbeiten” **32**
2.6 Prozess “Domänenwissen bearbeiten” **32**
2.7 Prozess “Informelle B-Anforderungen bearbeiten” **34**
2.8 Weiterführende Literatur **35**

KAPITEL 3 *System-Anforderungen bearbeiten* ***37***

3.1 Prozessverfeinerung **38**
3.2 Produkte **40**
B-Anforderungen ***41***
E-Anforderungen ***46***
Verfolgbarkeitsmatrix-Anforderungen ***57***
A-Verifikation ***58***
A-Validation ***62***
Namenskonventionen ***67***
3.3 Rollen **68**
3.4 Kontrollfluss **69**
3.5 Prozess “B-Anforderungen bearbeiten” **70**
3.6 Prozess “E-Anforderungen bearbeiten” **74**
3.7 Prozess “Verfolgbarkeitsmatrix-Anforderungen bearbeiten” **81**
3.8 Prozess “System-Anforderungen verifizieren” **83**
3.9 Prozess “Testfälle bearbeiten” **87**
3.10 Weiterführende Literatur **89**

KAPITEL 4 *System-Entwurf bearbeiten* ***91***

4.1 Prozessverfeinerung **93**
4.2 Produkte **95**
Entwurfsentscheidungen ***96***
UML-Entwurf ***98***
Verfolgbarkeitsmatrix-Entwurf ***110***
Komponenten-Anforderungen ***111***
E-Verifikation ***114***
E-Validation ***119***

Namenskonventionen ***126***
4.3 Rollen **126**
4.4 Kontrollfluss **128**
4.5 Prozess “Entwurfsentscheidungen erstellen” **129**
4.6 Prozess “UML-Entwurf bearbeiten” **130**
4.7 Prozess “Komponenten-Anforderungen erstellen” **139**
4.8 Prozess “Verfolgbarkeitsmatrix-Entwurf bearbeiten” **141**
4.9 Prozess “System-Entwurf verifizieren” **142**
4.10 Prozess “Komponenten-Testfälle bearbeiten” **146**
4.11 Weiterführende Literatur **148**

KAPITEL 5 *System-Komponenten bearbeiten* ***151***

5.1 Prozessverfeinerung **153**
5.2 Produkte **154**
Komponenten-Kode ***155***
K-Verifikation ***158***
Entwicklungsrichtlinien ***162***
Ausführbare Komponente ***166***
5.3 Rollen **169**
5.4 Kontrollfluss **170**
5.5 Prozess “Quellkode bearbeiten” **171**
5.6 Prozess “Ausführbare Komponente bearbeiten” **179**
5.7 Weiterführende Literatur **185**

KAPITEL 6 *System-Integration* ***187***

6.1 Prozessverfeinerung **188**
6.2 Produkte **191**
Ausführbares System ***192***
6.3 Rollen **199**
6.4 Kontrollfluss **200**
6.5 Prozess “Integriere Teilsystem” **201**
6.6 Prozess “Integrationstest” **204**

6.7 Weiterführende Literatur **207**

KAPITEL 7 *System-Validation* ***209***

7.1 Prozessverfeinerung **211**
7.2 Produkte **212**
Simulator ***213***
Reale Systemumgebung ***214***
7.3 Rollen **215**
7.4 Kontrollfluss **216**
7.5 Prozess “Systemtest” **217**
7.6 Prozess “Akzeptanztest” **220**
7.7 Prozess “Test im Betrieb” **223**
7.8 Weiterführende Literatur **226**

ANHANG A *Produkthierarchie* ***227***

ANHANG B *Prozesshierarchie* ***231***

ANHANG C *Kontrollfluss* ***235***

Index ***239***

KAPITEL 1

Einführung

In immer mehr technischen Produkten werden Steuerfunktionen von eingebauten Computersystemen (sog. *eingebetteten Systemen*) übernommen. Eingebettete Systeme sind heute in vielen Bereichen zu finden, z. B. in der Automobilindustrie (ABS, Motorsteuerung, etc.), im Konsumgütersektor (Digitalkameras, Mikrowellenherd, etc.), im medizinischen Bereich (Herzschrittmacher, künstliche Organe, etc.), in der Gebäudeautomation (Feuermeldesystem, Klimaanlage, Lichtsteuerung, etc.), in der Telekommunikation (Handy, Vermittlungsanlage, etc.), in der industriellen Produktion (Fertigungssteuerungen, Roboter, etc.) oder in der Flugzeugindustrie (Autopilot, Radarkontrolle, etc.). Die Bedeutung und Komplexität eingebetteter Systeme nimmt mit der zunehmenden Verbreitung und der Übernahme einer Vielzahl oftmals sicherheitskritischer Funktionen signifikant zu. Die Funktionalität eingebetteter Systeme wird immer weniger allein durch Hardware, sondern zunehmend durch Software (sog. *eingebettete Software*) realisiert. Dies erfordert geeignete Techniken, Methoden und Werkzeuge zur Entwicklung qualitativ hochwertiger Software für eingebettete Systeme. In diesem Bericht wird ein Entwicklungsprozess für die objektorientierte Entwicklung eingebetteter Software vorgestellt.

1.1 Zielsetzung

Ziel dieses Berichts ist die Dokumentation eines vollständigen Anwendungsentwicklungsprozesses für die Entwicklung von Software in eingebetteten Systemen. Dies beinhaltet alle Schritte von der Projektinitialisierung und Anforderungsfestlegung bis hin zur System-Validation.[1]

Dem Entwicklungsprozess liegen eine objektorientierte Vorgehensweise unter Verwendung der *Unified Modeling Language (UML*) sowie übliche Inspektions- und Testverfahren zugrunde. Der Prozess basiert auf Erfahrungen aus Projekten, in denen Software für eingebettete Systeme entwickelt wurde.

Im Einzelnen enthält dieser Bericht

- eine vollständige Beschreibung der Software-Entwicklungsaktivitäten mit allen eingesetzten Entwicklungstechniken und -methoden,
- eine vollständige Beschreibung der zugehörigen Entwicklungsprodukte (Dokumente und sonstige Artefakte, wie z. B. ausführbare Systemkomponenten) mit allen eingesetzten Beschreibungstechniken und -methoden,
- durchgängige, aufeinander abgestimmte, reale Beispiele für die Inhalte der Entwicklungsprodukte,
- ein Rollenkonzept, das Rollen (z. B. Anforderungsingenieur, Entwurfsingenieur, Validierer) definiert und den Rollen Entwicklungsaktivitäten zuweist,
- eine vollständige Baseline, d. h. die Charakterisierung des Entwicklungsprozesses hinsichtlich der wichtigen Kenngrößen Aufwand und Fehler im Kontext eines Projekts zur Entwicklung eines Gebäudeautomationssystems.

Dieses Dokument richtet sich hauptsächlich an

- *Software-Entwickler.* Die vorliegende Dokumentation kann als Anleitung zur Durchführung von Software-Entwicklungsaktivitäten verwendet werden. Es wird schrittweise erläutert, welche Aktivitäten wie durchzuführen sind. Es wird beschrieben, wie Aktivitäten mit anderen zusammenhängen und in welcher Reihenfolge Aktivitäten durchzuführen sind. Dabei wird aufgezeigt, welche

1. Nicht beschrieben wird die Entwicklung von Kommunikations- und Betriebssystemschichten.

Produkte benötigt bzw. erstellt werden und was der Inhalt dieser Produkte ist. Abgerundet wird die Dokumentation durch ein durchgängiges Beispiel, das einen Eindruck von den entstehenden Produkten und ihren Inhalten vermittelt.

- *Projektmanager und -planer,* die eine objektorientierte Methodik für die Entwicklung eingebetteter Software in ihrer Software-Entwicklungsorganisation etablieren möchten. Der hier vorgestellte Prozess mit der zugehörigen Dokumentationsstruktur kann als Grundlage genutzt werden. Der Prozess und die ihm zugrundeliegenden Konzepte wurden in mehreren Projekten erprobt und haben sich bewährt, beanspruchen jedoch keine Allgemeingültigkeit für die Software-Entwicklung im Bereich “Eingebetteter Systeme”. Vielmehr bedarf es Anpassungen an konkrete Aufgabenstellungen sowie an den System-, den Projekt- und den Organisationskontext. In diesem Bericht angegebene Aufwands- und Fehlerverteilungen können unter Berücksichtigung des Projektkontexts als Ausgangspunkt für die Aufwands- und Qualitätssicherungsplanung verwendet werden.
- *Qualitätssicherer.* Der hier vorgestellte Prozess beinhaltet eine Methodik zur systematischen Verifikation[1] und Validation[2] im Kontext objektorientierter Entwicklung eingebetteter Software. Es wird beschrieben, zu welchen Zeitpunkten Verifikations- und Validierungsaktivitäten durchgeführt werden können und wie deren Ergebnisse dokumentiert werden. Es wird aufgezeigt, wie Verifikations- und Validationsaktivitäten genutzt werden können, um möglichst frühzeitig Fehler zu finden und zu korrigieren.

Voraussetzung für das Verständnis dieses Berichts sind Erfahrungen mit der Entwicklung von Software sowie Grundkenntnisse in UML. Bei der Beschreibung des Entwicklungsprozesses wurde angestrebt, bekannte Schwächen existierender Software-Entwicklungsstandards und Beschreibungen von Vorgehensmodellen zu beheben. Solche Beschreibungen sind vielfach gekennzeichnet durch

- den Anspruch, für ein sehr breites Anwendungsspektrum geeignet zu sein (z. B.: Eignung sowohl für interaktive Systeme als auch für transformationelle und eingebettete Systeme[3]). Angaben zur tatsächlichen Eignung der Vorgehensmodelle für diese verschiedenen Domänen und hiermit verbundene Anpas-

1. Verifikation: alle Aktivitäten, die der Überprüfung der Korrektheit eines Softwaredokuments (z. B. UML-Entwurf oder Komponenten-Kode) dienen.
2. Validation: alle Aktivitäten, die der Überprüfung der Zuverlässigkeit von ausführbaren Produkten (z. B. ausführbares Teilsystem oder System) dienen.

sungsvorschläge fehlen jedoch oftmals gänzlich oder sind nur rudimentär vorhanden. Dies lässt häufig auf eine geringe Erprobung der Prozesse in verschiedenen Domänen schließen.

- die Behauptung, effiziente Softwareentwicklung zu ermöglichen. Es existieren jedoch oftmals keinerlei Nachweise (z. B. in Form von Baselines), die dies quantitativ untermauern würden.
- abstrakte Beschreibungen. Es fehlen häufig detaillierte Vorgehens- und Dokumentbeschreibungen sowie durchgängige Beispiele.
- einen hochgradig iterativen Prozess. Dies spiegelt meist nur die geringe Erfahrung mit dem Prozess wider und erschwert eine systematische Kontrolle des Projektablaufs.

1.2 Charakterisierung des Entwicklungsprozesses

Der hier vorgestellte Prozess lässt sich charakterisieren als

- domänenspezifisch,
- objektorientiert,
- verhaltensorientiert,
- qualitätsorientiert und
- kontrolliert iterativ.

Es wird ein Ersterstellungsprozess beschrieben. Teile des Prozesses können auch im Rahmen der Wartung von eingebetteten Systemen angewandt werden. Dies erfordert jedoch eine sorgfältige Integration des hier beschriebenen Ersterstellungsprozesses mit dem Wartungsprozess.

3. Eingebettete Systeme lassen sich einerseits gegenüber *transformationellen Systemen* abgrenzen, die zu gegebenen Eingabedaten ein Endresultat berechnen, um dann in einen Terminierungszustand überzugehen. Auf der anderen Seite lassen sich eingebettete Systeme gegenüber *interaktiven Systemen* abgrenzen, die zwar auch mit ihrer Umgebung in Wechselwirkung stehen, aber nur in dem vom System vorgegebenen Takt. Eine terminologische Abgrenzung der Begriffe “transformationelles System” und “interaktives System” findet sich in Halbwachs (1993).

Domänenspezifisch

Der vorgestellte Entwicklungsprozess ist für Software in eingebetteten Systemen ausgelegt. Eingebettete Systeme sind durch Hardware gekapselt und interagieren stark mit dem physikalischen System in ihrer Umgebung. Sie müssen 1) Informationsquellen beobachten können, um Entwicklungen der Umwelt zu erkennen; 2) vorbereitet sein auf zufällig eintretende Ereignisse; 3) fähig sein, Entscheidungen zu treffen, die auf Ereignissen und der Beobachtung beruhen; 4) in der Lage sein, Aktionen in der Umwelt zu veranlassen.

Vier wesentliche Charakteristika eingebetteter Software seien an dieser Stelle hervorgehoben, da sie auf den in diesem Bericht beschriebenen Entwicklungsprozess signifikanten Einfluss ausüben:

1. *Eingebettete Software muss (weitgehend) fehlerfrei sein.* Dies hat mehrere Ursachen: Zum einen erwartet der spätere Benutzer ein korrekt arbeitendes System, zum anderen ist der Austausch der Software oftmals nicht möglich und somit kann auch keine Software-Wartung stattfinden.
   Einfluss auf den Entwicklungsprozess: Aus den hohen Anforderungen an die Korrektheit resultiert die Qualitätsorientierung des Entwicklungsprozesses. Hohe Qualität eingebetteter Software wird insbesondere durch frühzeitige Erkennung bzw. systematische Vermeidung von Fehlern erzielt.
2. *Nicht-funktionale Eigenschaften (z. B. Echtzeitfähigkeit, Zuverlässigkeit, Fehlertoleranz) haben eine besondere Bedeutung.* Dies hat seine Ursache vor allem in den Anwendungsgebieten von eingebetteten Systemen. Echtzeitfähigkeit ist beispielsweise in der Robotersteuerung von großer Bedeutung. Zuverlässigkeit und Fehlertoleranz spielen eine große Rolle bei sicherheitskritischen Anwendungen wie Flugzeugsteuerungen.
   Einfluss auf den Entwicklungsprozess: Der besonderen Bedeutung nicht-funktionaler Anforderungen trägt der Entwicklungsprozess durch eine gesonderte Behandlung solcher Anforderungen Rechnung (z. B. durch spezielle Beschreibungsformen).
3. *Die Softwareentwicklung setzt ein genaues Verständnis der umgebenden Hardware voraus.* Dies resultiert aus der Integration eingebetteter Software in ein sie umgebendes Hardware-System, das vielfach anwendungsspezifisch ist. Das Vorhandensein offener Plattformen ist bei eingebetteten Systemen (noch) unüblich.
   Einfluss auf den Entwicklungsprozess: Aus der Integration eingebetteter Software in ein Hardware-System und der starken Interaktion eingebetteter Software mit der physikalischen Umgebung resultiert insbesondere die Verhaltensorientierung des Prozesses.

4. *Wirtschaftliche Aspekte wie Entwicklungskosten und -zeit spielen eine große Rolle.* Der Markt für eingebettete Systeme ist einer der am stärksten wachsenden Märkte im Informations- und Telekommunikationssektor. Hieraus resultiert die Forderung nach immer kürzer werdenden Produktzyklen sowie kosteneffizienteren Entwicklungs- und Produktionsverfahren.
   Einfluss auf den Entwicklungsprozess: Aus den wirtschaftlichen Anforderungen resultiert die besondere Beachtung der Plan- und Kontrollierbarkeit des Prozesses, die mittels kontrollierter Iterationen erreicht wird.

Objektorientiert

Der Entwicklungsprozess verwendet zur Modellierung objektorientierte Konzepte. Dies sind:

Datenabstraktion und Kapselung: Das Konzept basiert auf dem des abstrakten Datentyps. Zusammengehörige Daten (Attribute) und Funktionen (Methoden) werden in Objekten gekapselt. Die Methoden operieren auf den Attributen, die den Zustand eines Objekts bilden. Anderen Objekten ist nur die Dienstschnittstelle, die von den öffentlichen Methoden eines Objekts gebildet wird, bekannt.

Klassen: Eine Menge von Objekten gleichen Verhaltens und gleicher Struktur wird durch die Abstraktion “Klasse” beschrieben. In einem objektorientierten Programm existieren Klassen nur im Programmtext und Objekte als Instanzen dieser Klassen nur zur Laufzeit.

Vererbung: Attribute und Methoden werden von der übergeordneten Klasse an die entsprechenden Unterklassen vererbt. Die Unterklassen können um weitere Funktionalität ergänzt werden.

Kunde-Lieferant-Beziehung: Zu Erfüllung einer Methode ist ein Objekt häufig auf andere angewiesen. Das Kunden-Objekt benutzt dann die benötigten Methoden der Lieferanten-Objekte.

Durch die Verwendung objektorientierter Konzepte soll die Einhaltung von Modularisierungsprinzipien, wie dem Kopplungs- und Geheimnisprinzip, erleichtert werden. Eine angemessene Modularisierung unterstützt wiederrum die Verständlichkeit, Änderbarkeit und Wiederverwendbarkeit eines Systems. Die objektorientierten Konzepte werden durchgängig von der Analyse bis zur Implementierung verwendet. Ein Übergang zwischen den Entwicklungsschritten ist ohne Paradigmenwechsel möglich.

Verhaltensorientiert

Der Entwicklungsprozess legt einen Schwerpunkt auf die Beschreibung des Verhaltens des Systems. Dies liegt einerseits an der komplexen Interaktion von eingebetteter Software mit ihrer Umgebung. Andererseits ist die Komplexität eingebetteter Systeme ursächlich in ihren dynamischen Verhalten begründet (und weit weniger in ihrer Architektur). Die Verhaltensorientierung des Prozesses spiegelt sich beispielsweise in der umfassenden Modellierung von Zustands-, Sequenz und Kollaborationsdiagrammen während der Anforderungsanalyse wider.

Qualitätsorientiert

Der Prozess sieht an wichtigen Meilensteinen Verifikations- und Validationsaktivitäten vor, die für eingebettete Software maßgeschneidert sind (z. B. durch die Nutzung domänenspezifischer Checklisten). Ziel ist, Fehler möglichst frühzeitig zu erkennen und zu entfernen. Um eine möglichst effektive und effiziente Fehlererkennung und -beseitigung zu unterstützen, sieht der Prozess eine systematische Dokumentation sämtlicher Entwicklungsdokumente (auf allen Abstraktionsstufen) vor. Die Verfolgbarkeit dieser Dokumente wird durch matrixartige Tabellen beschrieben. Dies erleichtert eine konsistente Durchführung von Änderungen. Die Qualitätsorientierung des Prozesses spiegelt sich beispielsweise in der Bereitstellung von Dokumentationsvorlagen und domänenspezifischen Checklisten wider. Der Prozess unterstützt die Entwicklung einer verständlichen und leicht änderbaren System-Dokumentation.

Kontrolliert iterativ

Bei eingebetteten Systemen steht die grobe Funktionalität relativ früh im Entwicklungsprozess fest (z. B. Licht- und Temperaturregelung bei Gebäudeautomationssystemen). Somit kann in der Regel eine frühzeitige Aufteilung des Systems in Teilsysteme erfolgen. Dies kommt der Anforderung nach einem planbaren und kontrollierbaren Entwicklungsprozess entgegen. Der hier vorgestellte Prozess erlaubt Iterationen nur zu fest definierten Zeitpunkten. Die Iterativität des Prozesses zeigt sich beispielsweise in der Anbindung von Rework-Aktivitäten im Anschluss an Verifikations- und Validationsschritte. Der Entwicklungsprozess erlaubt darüber hinaus eine inkrementelle Entwicklung der Software mit einer anschließenden, gemeinsamen Integration der Inkremente.

1.3 Herkunft des Prozesses

Der hier vorgestellte Entwicklungsprozess wurde aufbauend auf folgenden Quellen entwickelt:

- einem im *Sonderforschungsbereich 501* entwickelten Referenzprozess zur Entwicklung großer Systeme;
- Erfahrungen aus Software-Entwicklungsprojekten zur Erstellung von Gebäudeautomationssystemen, die von der *AG Software Engineering* und dem Sonderforschungsbereich 501 an der Universität Kaiserslautern durchgeführt wurden;
- dem zur Software-Entwicklungsmethode *Object Modeling Technique (OMT)* gehörenden Software-Entwicklungsprozess. OMT ist eine der am meisten verbreiteten Methoden im Bereich der objektorientierten Anwendungsentwicklung, insbesondere beim Einsatz für objektorientierte Analyse und Entwurf. Basierend auf dem *Unified Process* wurden Verbesserungen, Erweiterungen und Ergänzungen vorgenommen.

1.4 Prozessbeschreibung

Im Folgenden wird erläutert, nach welchem Schema und auf welche Art und Weise der Softwareentwicklungsprozess in diesem Bericht dokumentiert ist. Hierbei werden insbesondere die Begriffe Produkt, Prozess und Rolle genauer definiert.

Produkte

Als Produkte (synonym: Artefakte) werden alle Ergebnisse von Software-Entwicklungsaktivitäten sowie Informationen, die für deren Durchführung benötigt werden, bezeichnet. Beispiele für Produkte sind ein Anforderungsdokument, eine Entwurfsdokument, ein UML-Klassendiagramm, ein Kodestück oder eine ausführbare Kodekomponente. Produkte können als Verfeinerung eine Menge von Teilprodukten besitzen, die wiederum verfeinert sein kann. So kann beispielsweise ein UML-Entwurf aus einer Menge von UML-Diagrammen unterschiedlicher Typen bestehen. Dieser Bericht enthält eine vollständige Produktstruktur für die Produkte eines Software-Systems. Es wird eine Verfeinerungshierarchie angegeben und beschrieben, welche Inhalte in den Produkten enthalten sind. Abbildung 1 zeigt die grafische Darstellung von Produkten anhand eines Beispiels. Hier ist das Produkt Problemlösungsdokumentation in drei Teilprodukte (Problembeschreibung, Anfor-

derungen und Validationsdokument) verfeinert. Das kleine Quadrat im Produkt Anforderungen deutet an, dass dieses Produkt wiederum aus einer Menge von Teilprodukten besteht. Im Anhang ist eine Übersicht über die gesamte Produkthierarchie enthalten. Produkte lassen sich noch weiter strukturieren hinsichtlich ihrer inhaltlichen Gliederung. Eine solche inhaltliche Strukturierung ist oftmals feingranularer als die Produktverfeinerungshierarchie. Sowohl die Produkthierarchie als auch die inhaltliche Strukturierung einzelner Teilprodukte werden in diesem Bericht übersichtlich mit UML-Packagediagrammen dargestellt. In diesem Bericht gibt es für die Inhalte der einzelnen Produkte ausführliche textuelle Erläuterungen mit Beispielen.

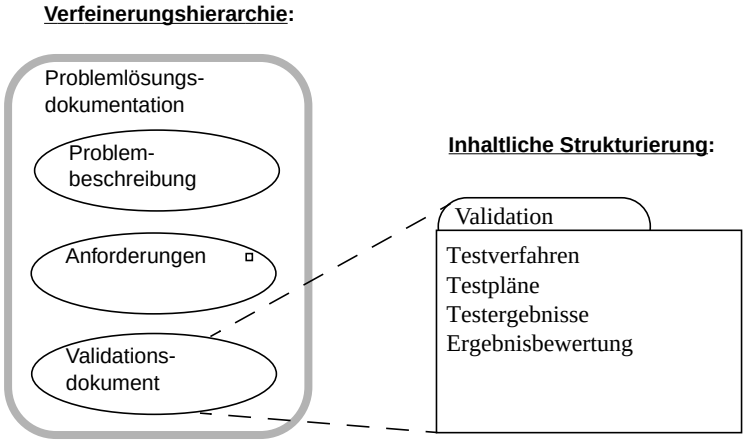


ABBILDUNG 1. Repräsentation von Produkten

Einige Produkte (z. B. Kodekomponente, Teilsystem etc.) können in mehrfacher Ausführung vorhanden sein. Dies ist im Kontext der jeweiligen Beschreibung erkennbar. Die hier eingeführte Produktstruktur muss sich nicht unmittelbar auf Dateiebene widerspiegeln; eine eindeutige Zuordnung von Produkten und Teilprodukten zu Dateien ist jedoch erforderlich. In diesem Bericht werden die Produkte jeweils in einem größeren Zusammenhang *vor* den zugehörigen Entwicklungsaktivitäten beschrieben, da zum einen Entwickler oftmals sehr viel mehr mit Entwicklungsprodukten vertraut sind als mit deren Erstellungs- bzw. Bearbeitungsprozeduren und zum anderen die Beschreibung der Entwicklungsaktivitäten eine Kenntnis der Produkte voraussetzt.

Prozesse

Ein Prozess (synonym: Aktivität) beschreibt eine Menge partiell geordneter Schritte, um ein Ziel zu erreichen. Beispiele für Prozesse sind das Erstellen der System-Anforderungen, die Bearbeitung des Entwurfs, die Verifikation von Kodekomponenten, die Integration von Teilsystemen oder ein Akzeptanztest. Prozesse können als Verfeinerung eine Menge von Teilprozessen besitzen, die wiederum verfeinert sein können. Im Anhang ist eine Übersicht über die gesamte Prozesshierarchie enthalten. Prozesse sind gegenstandslose Phänomene der realen Welt. Deshalb werden in diesem Bericht grafische Repräsentationen zur Darstellung von Prozessen benutzt. Abbildung 2 zeigt beispielsweise den Prozess “Problembeschreibung erstellen”. Für diesen Prozess werden die Teilprodukte Systemumgebung und Referenzdokumente (beides Teilprodukte des Domänenwissens) benötigt. Resultat des Prozesses ist eine Problembeschreibung, die Teil der Problemlösungsdokumentation ist. Prozessverfeinerungen werden analog zu Produktverfeinerungen dargestellt. Manche Prozesse können mehrfach instantiiert werden. So kann es beispielsweise für jede Systemkomponente einen Prozess zu ihrer Implementierung geben. Die Beziehungen zwischen Prozessen und Produkten (auch Produktfluss genannt) lassen sich folgendermaßen unterscheiden: 1) ein *konsumierender Produktfluss* bedeutet, dass ein Produkt in einem Prozess gelesen werden kann, 2) ein *produzierender Produktfluß* bedeutet, dass ein Produkt in einem Prozess erstellt wird (schreibender Zugriff) und 3) ein *modifizierender Produktfluss* bedeutet, dass auf ein Produkt sowohl schreibend als auch lesend zugegriffen werden darf. Die grafischen Darstellungen vermitteln eine grobe Beschreibung der erlaubten Produktflüsse. Detailliertere Beschreibungen befinden sich jeweils in den Erläuterungen zu den einzelnen Prozessen.

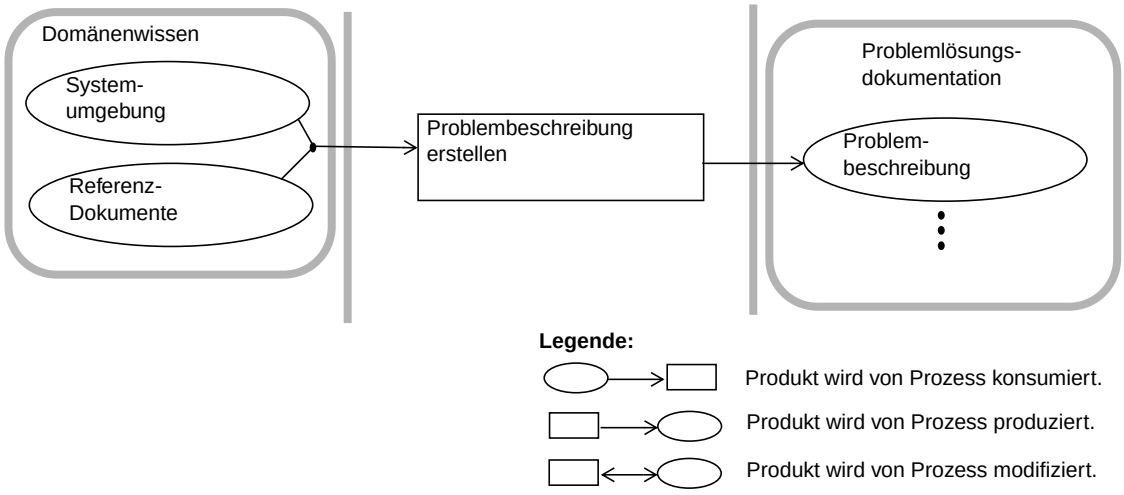


ABBILDUNG 2. Repräsentation von Prozessen

Der Produktfluß sagt zunächst nichts aus über die Reihenfolge der Durchführung der Prozesse. Die Durchführungsreihenfolge (auch Kontrollfluss genannt) wird in diesem Bericht mit Präzedenzgraphen veranschaulicht. Ein Präzedenzgraph beschreibt die Abhängigkeiten der Prozesse hinsichtlich der Abarbeitungsfolge. Ein Beispiel ist in Abbildung 3 gegeben. Die Pfeile geben die Reihenfolge der Abarbeitung vor. Eine Tätigkeit kann erst begonnen werden, sobald alle Tätigkeiten mit eingehendem Pfeil beendet sind. Genauere Angaben zum Kontrollfluss befinden sich in der textuellen Beschreibung. Eine Gesamtübersicht über die Abarbeitungsfolge aller Prozesse findet sich im Anhang.

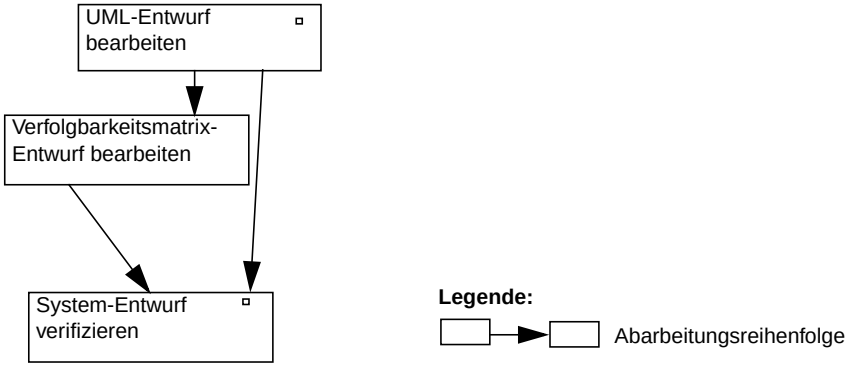


ABBILDUNG 3. Repräsentation der Abarbeitungsreihenfolge

Rollen

Software-Entwicklungsprozesse werden von Agenten (Personen oder Werkzeugen) durchgeführt. Diese spielen bei der Durchführung eines Prozesses bestimmte Rollen. Eine Rolle trägt die Verantwortung für die Erledigung der mit dem Prozess verbundenen Aufgaben. Beispiele für Rollen sind der Anforderungsingenieur, der Entwurfsingenieur, der Validierer oder der Kunde. Eine Rolle kann von mehreren Agenten gespielt werden; ein Agent kann auch mehrere Rollen spielen. In manchen Fällen sind Rollen direkt für Produkte verantwortlich. Abbildung 4 zeigt, wie Rollen und ihre Produktverantwortlichkeiten in diesem Bericht grafisch dargestellt werden.

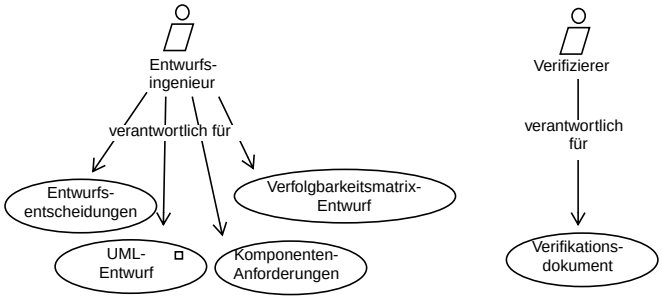


ABBILDUNG 4. Repräsentation von Rollen

1.5 Der Prozess im Überblick

Abbildung 5 gibt einen Überblick über den Gesamtprozess. In Anlehnung an zwei wesentliche Eigenschaften (Domänenorientierung und Iterativität) wird der Prozess als *Do-it-Prozess* bezeichnet. Es werden die Hauptprozesse auf der obersten Abstraktionsstufe gezeigt sowie die wesentlichen Produktflüsse. Zur deutlicheren Darstellung des Gesamtprozesses werden *nicht alle* Produktflüsse dargestellt. In der Abbildung sind jeweils nur diejenigen Rollen gezeigt, die an der Durchführung der Prozesse *wesentlich* beteiligt sind. Tatsächlich sind jeweils eine Reihe weiterer Rollen involviert. Software wird entwickelt und nicht produziert. Somit ist auch jeder Prozess ein Unikat. Der Do-it-Prozess ist nicht als allgemeingültiger Prozess für alle Software-Entwicklungsprojekte in der Domäne “Eingebettete Systeme” anzusehen. Vielmehr sollte er als Rahmen angesehen werden, der jeweils an den Projekt-, den Organisations- und den Systemkontext angepasst werden muss.

Der vorliegende Bericht gliedert sich entlang der Beschreibungen der Hauptprozesse.

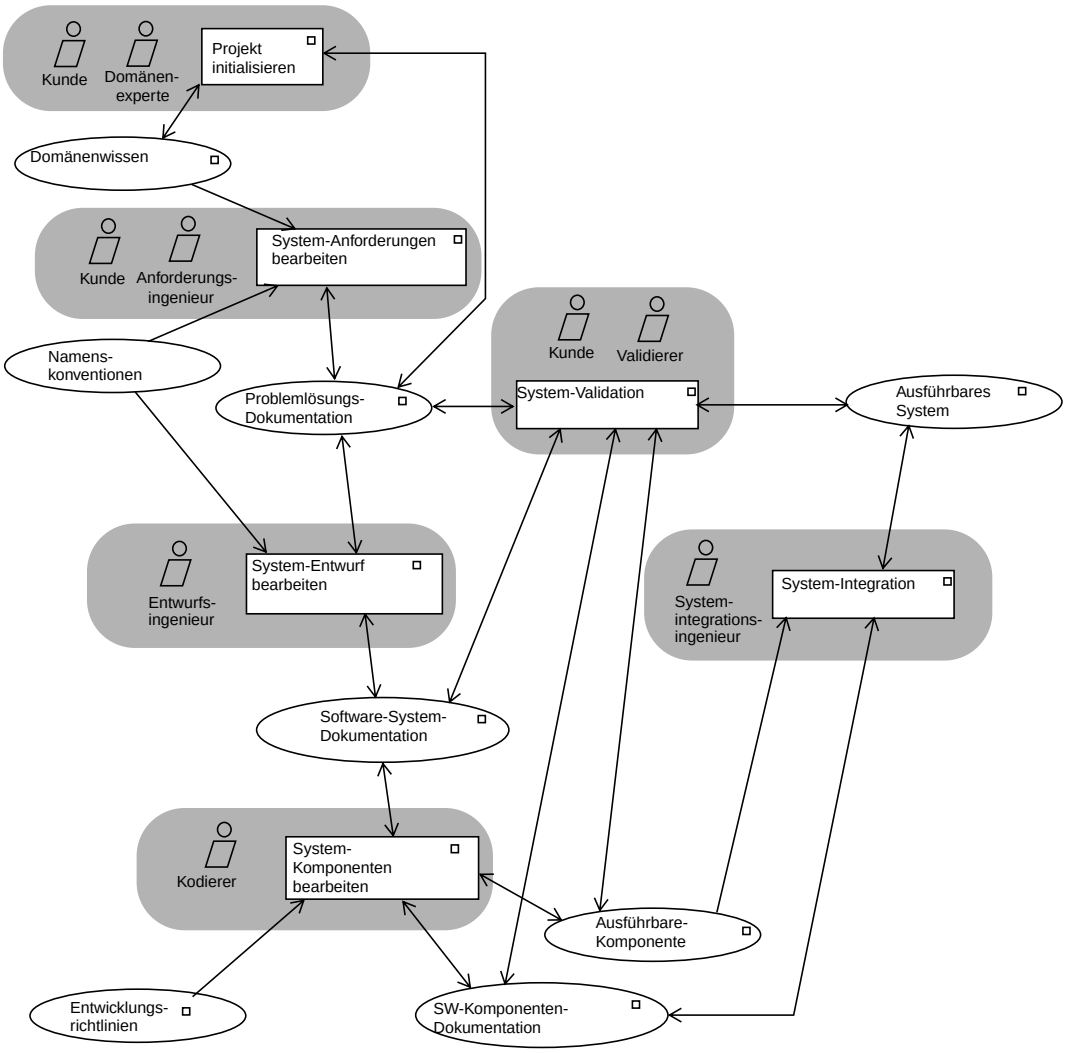


ABBILDUNG 5. Der Gesamtprozess

1.6 Das Anwendungsbeispiel

Der Bericht ist durchgängig mit Beispielen für Inhalte von Produkten versehen. Diese Beispiele stammen alle aus einem Anwendungsprojekt. Dieses Projekt sowie die dabei verwendeten Werkzeuge sollen hier kurz charakterisiert werden. Im Rahmen des Projekts wurde ein Gebäudeautomationssystem mit einer Licht- und Temperaturregelung erstellt. Hierbei wurden als wesentliche Techniken UML, C++, Checklisten-orientierte Inspektionen und Black-Box-Testen eingesetzt. Die Entwicklung erfolgte entsprechend des Do-it-Prozesses. Das Projekt wurde von 17 Software-Entwicklern (Studenten mit Software-Engineering-Kenntnissen) an der Universität Kaiserslautern im Jahr 1999 durchgeführt. Das resultierende Software-System hat einen Umfang von 22 KLOC. Die Produktivität betrug 8 LOC/Stunde[1]. Als Werkzeug zur Modellierung und (Teil-)Generierung der UML-Diagramme wurde Software through Pictures (StP) von Aonix in der UML-Version verwendet. Die Dokumentation erfolgte weitgehend mit dem Textverarbeitungssystem FrameMaker. Große Teile der Dokumentation konnten automatisch generiert werden, so dass ein einfaches Zusammenspiel von StP und FrameMaker möglich war. Hierfür wurde ein eigenentwickelter Generator (Wagenbichler, 1999) eingesetzt. Zur Übersetzung von Kode und zur Systemintegration wurden UNIX-übliche Werkzeuge verwendet.

1.7 Planung

Im Folgenden werden eine Aufwands- und Fehlerverteilung des Do-it-Prozesses angegeben, die bei der Durchführung des oben beschriebenen Anwendungsprojekt ermittelt wurden. Ausgehend von einer Schätzung des Gesamtaufwandes kann die Aufwandsverteilung als Ausgangspunkt für eine Planung der Aufwände genutzt werden, wobei zu beachten ist, dass Unterschiede im Projektkontexts (z. B. andere Anwendungsdomäne, andere Entwicklererfahrungen, andere Entwicklungswerkzeuge) Einfluss auf die Beschaffenheit der Aufwands- und Fehlerverteilungen haben. Insofern können die angegebenen Verteilungen nur als Richtschnur angesehen werden. Die Fehlerverteilung kann helfen, ungewöhnliche Fehlerhäufungen in einzelnen Dokumenten zu erkennen. Es ist wiederum der Projektkontext zu berücksichtigen.

1. LOC steht für Lines of Code (Programmzeilen, hier: inklusive Kommentarzeilen)

Mehrere Projekte, die in der Domäne “Hausautomatisierung” durchgeführt wurden, haben gezeigt, dass die im Folgenden dargestellten Aufwands- und Fehlerverteilungen durchaus typisch für den Do-it-Prozess in dieser Domäne sind.

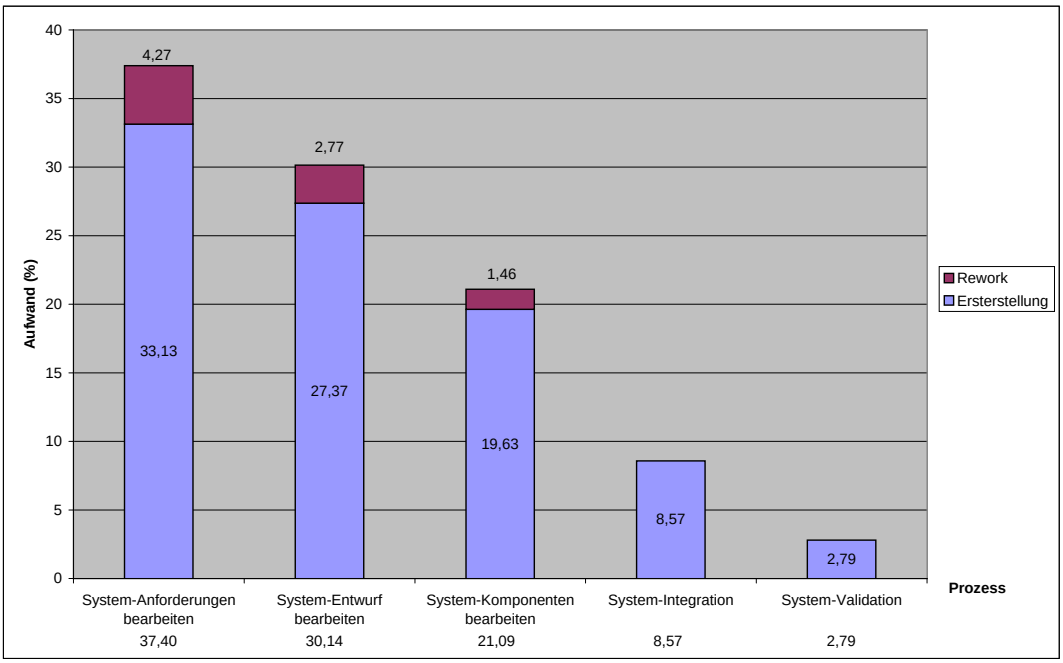


ABBILDUNG 6. Aufwandsverteilung

Das *Aufwandsmodell* in Abbildung 6 zeigt eine (37%, 30%, 21%, 9%, 3%)-Verteilung auf die Phasen Analyse, Entwurf, Kodierung (mit integrierter Komponenten-Validation), Integration und System-Validation. Besonders fällt auf, dass

- der Aufwand für die System-Validation außergewöhnlich gering (3%) ist. In industriellen Projekten geht man i. a. von bis zu 50% aus. Der geringe Anteil kann auf die frühzeitige Entdeckung vieler Fehler bei Inspektionen zurückgeführt werden.
- der Aufwand in den frühen Prozessschritten (Analyse und Entwurf) verhältnismäßig hoch (67%) ist. Dies ist im wesentlichen auf zwei Analyseschritte und die umfangreiche Modellierung verschiedener UML-Sichten zurückzuführen.

Die Produktivität bezogen auf den Gesamtaufwand lag im Anwendungsprojekt jedoch mit 8 LOC/h (Lines of Code/Stunde) weit über dem Durchschnitt industrieller Projekte (5 LOC/h für Organisationen mit CMM-Level 3).

Das *Fehlerslippagemodell* in Abbildung 7 zeigt eine Verteilung der Fehler auf Produkte. Es ist nach den Produkten, in denen die Fehler auftreten, gegliedert und gibt Auskunft über den jeweiligen Zeitpunkt der Entdeckung eines Fehlers. Besonders fällt auf, dass die meisten der besonders teuren Anforderungsfehler (96%) bereits frühzeitig bei der Anforderungsinspektion entdeckt wurden.

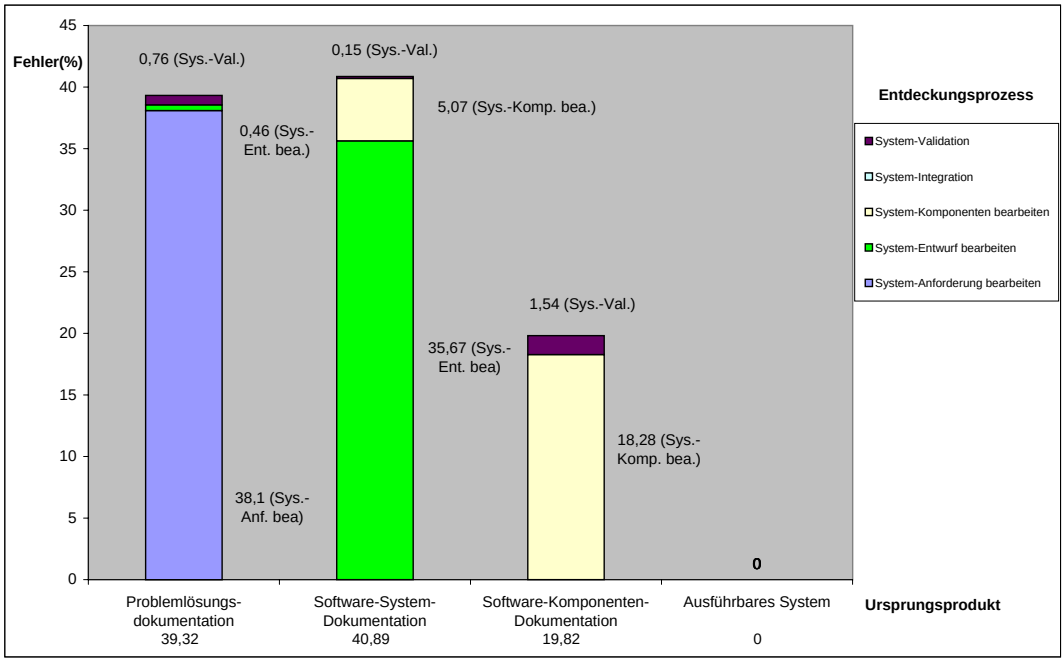


ABBILDUNG 7. Fehlerverteilung

Das dargestellte Fehlerslippagemodell zeigt die Verteilung der Fehler über die Problemlösungsdokumentation (diese umfasst die Analyse), die Software-System-Dokumentation (diese umfasst den Entwurf), die Software-Komponenten-Dokumentation (diese umfasst die Komponenten-Entwürfe und den Kode) sowie das ausführbare System. Die Prozesse ‘System-Anforderungen bearbeiten’, ‘System-

Entwurf bearbeiten' und 'System-Komponenten bearbeiten' umfassen jeweils eine abschließende Inspektion der erstellten Dokumente. Die meisten Fehler, die während der Analyse gemacht wurden, konnten während der Anforderungsinspektion entdeckt werden. Nur 0,46% der Analysefehler wurden erst in der Entwurfsphase und nur 0,76% bei der System-Validation gefunden. Dies spricht für eine sehr effektive Anforderungsinspektion und einen hohen Reifegrad der verwendeten Checklisten. Entwurfsfehler konnten nicht in diesem Umfang bei der Entwurfsverifikation entdeckt werden: 5,23% wurden erst in späteren Phasen entdeckt. Eine Analyse ergab, dass vor allem auf Inkonsistenzen beruhende Fehler (24%) sowie Unvollständigkeitsfehler (48%) im Entwurfsdokument vorherrschten. Dies ist auf eine unzureichende horizontale Verfolgbarkeit zwischen verschiedenen UML-Entwurfssichten zurückzuführen. Bei der Komponentenentwicklung sind ca. 20% der Fehler aufgetreten, jedoch konnten die meisten (18,28%) bei der Kodeinspektion entdeckt und daraufhin behoben werden. Da das Modell die Fehler entsprechend ihres Ursprungsprodukts zuordnet, sind dem ausführbaren System keine Fehler zugeordnet. Bei der System-Validation entdeckte Fehlverhalten werden im Modell auf Fehler in den jeweiligen Dokumentationen abgebildet.

1.8 Weiterführende Literatur

1. Aonix: *Software through Pictures: User Manuals*. Aonix Release 2.4 und 3.4, 1998.
2. Jean Paul Calvez: *Embedded Real-Time Systems: A Specification and Design Methodology.* Wiley, 1993.
3. Nicolas Halbwachs: *Synchronous Programming of Reactive Systems*, Kluwer Academic Publisher, 1993.
4. Ivar Jacobson, Grady Booch, James Rumbaugh: *Unified Software Development Process*. Addison Wesley Publishing Company, 1999.
5. Ivar Jacobson and Grady Booch and James Rumbaugh: *The Unified Process.* IEEE Software, Vol. 16, Nr. 3, S. 96-102, Mai 1999.
6. Richard A. Johnson, B. C. Hardgrave: *Object-oriented methods: current practices and attitudes.* Journal of Systems and Software, Vol. 48, Nr. 1, S. 5-12, August 1999.
7. Jürgen Münch: *Anpassung von Vorgehensmodellen im Rahmen ingenieurmäßiger Softwarequalitätssicherung*. Im Tagungsband des 6. Workshops der Fachgruppe 5.1.1 (GI): Vorgehensmodelle, Prozessverbesserung und Qualitätsmanagement, Kaiserslautern, 19.-20. April, 1999.

8. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen: *Object-Oriented Modeling and Design.* Prentice Hall, 1990.
9. Martin Verlage, Jürgen Münch: *Formalizing Software Engineering Standards.* Proceedings of the Third International Symposium and Forum on Software Engineering Standards (ISESS'97), California, USA, June 1-6, 1997.
10. Wolfgang Wagenbichler: *Erstellung einer SW-Systemdokumentation mit StP.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 67653 Kaiserslautern, 1999. URL: http://wwwagse.informatik.uni-kl.de/access-system/publications.html#8

KAPITEL 2

Projekt initialisieren

Ziel des Prozesses *Projekt initialisieren* ist, ein Entwicklungsprojekt für ein eingebettetes Software-System aufzusetzen sowie die wichtigsten Anforderungen an das zu entwickelnde System und Ziele für das durchzuführende Projekt aufzustellen.

Es gibt verschiedene Gründe, warum ein Entwicklungsprojekt für ein eingebettetes Software-System aufgesetzt wird. Beispielsweise identifiziert ein Kunde Probleme in seiner Umgebung, die durch die Einführung eines Software-Systems behoben werden können (z. B. überhöhter Energiebedarf in einem Universitätsgebäude). Probleme, die ein eingebettetes Software-System lösen kann, lassen sich z. B. durch Marktforschung und -analyse ermitteln. Neben konkreten Problemen können auch strategische Ziele zum Aufsetzen eines Software-Entwicklungsprojekts führen.

Wurde der Bedarf für ein Software-System festgestellt[1], werden die wichtigsten Anforderungen an das System zur Lösung der identifizierten Probleme oder zum Erreichen der strategischen Ziele aufgestellt. Für die Formulierung dieser ersten Anforderungen sind genauere Kenntnisse der Anwendungsdomäne (z. B. Wissen

1. An dieser Stelle wird der Entwicklungsprozess idealisiert dargestellt. In vielen Fällen wird ein Software-System nicht völlig neu (“auf der grünen Wiese”) entwickelt. Es müssen existierende Software-Systeme erweitert werden oder existierende Software-Systeme dienen als Referenzentwicklungen.

über das Gebiet Gebäudeautomatisierung) notwendig. Solches Wissen wird im Folgenden als Domänenwissen bezeichnet.

Abbildung 8 gibt einen Überblick über die verwendeten Produkte im Prozess *Projekt initialisieren*.

Folgende Produkte werden modifiziert:

- *Domänenwissen*. Dieses Dokument beschreibt typische Eigenschaften und Charakteristika der Anwendungsdomäne. Es umfasst insbesondere eine Beschreibung der Systemumgebung.
- *Problemlösungsdokumentation*. Dieses Dokument umfasst eine Problembeschreibung, in der der Bedarf für das zu entwickelnde eingebettete Software-System erläutert wird. Das Dokument enthält außerdem eine Beschreibung der Anforderungen aus Sicht des Benutzers/Kunden und eine Beschreibung der Anforderungen aus Sicht der Entwickler. Außerdem umfasst es eine Verfolgbarkeitsmatrix, mit deren Hilfe Zusammenhänge innerhalb der System-Anforderungen explizit gemacht werden. Des Weiteren enthält die Problemlösungsdokumentation Dokumente zu Verifikations-[1] und Validationsaktivitäten[2].

ABBILDUNG 8. Verwendete Produkte des Prozesses *Projekt initialisieren*

2.1 Prozessverfeinerung

Der Prozess *Projekt initialisieren* unterteilt sich in drei Teilprozesse (siehe Abbildung 9). Der Bedarf für ein eingebettetes System wird ermittelt. Dies kann beispielsweise durch einen Kunden geschehen, der seine Umgebung beobachtet und

1. Verifikationsaktivitäten sind Tätigkeiten, die der Überprüfung der Korrektheit eines Softwaredokuments (z. B. UML-Entwurf oder Komponenten-Kode) dienen.
2. Validationsaktivitäten sind Tätigkeiten, die der Überprüfung der Zuverlässigkeit von ausführbaren Produkten (z. B. ausführbares Teilsystem oder System) dienen.

die existierende Situation analysiert. Es werden Probleme identifiziert, die durch ein eingebettetes System gelöst werden können oder es werden strategische Ziele definiert, die durch ein eingebettetes System erreicht werden können. Aus dem Bedarf heraus werden die wichtigsten Anforderungen an die Funktionalität des Systems abgeleitet. Der Bedarf wird in der *Problembeschreibung* motiviert (dies geschieht im Prozess *Problembeschreibung bearbeiten*). Zur Unterstützung der weiteren System-Entwicklung wird *Domänenwissen* dokumentiert (im Prozess *Domänenwissen bearbeiten*). Dafür wird einerseits die *Systemumgebung* beschrieben. Andererseits wird überprüft, ob bereits ähnliche Projekte initialisiert wurden. Ist dies der Fall, werden Erfahrungen aus diesen Projekten in *Referenz-Dokumenten* bereitgestellt. Im Prozess *Informelle B-Anforderungen bearbeiten* werden die Anforderungen aus der *Problembeschreibung* konkretisiert und erweitert.

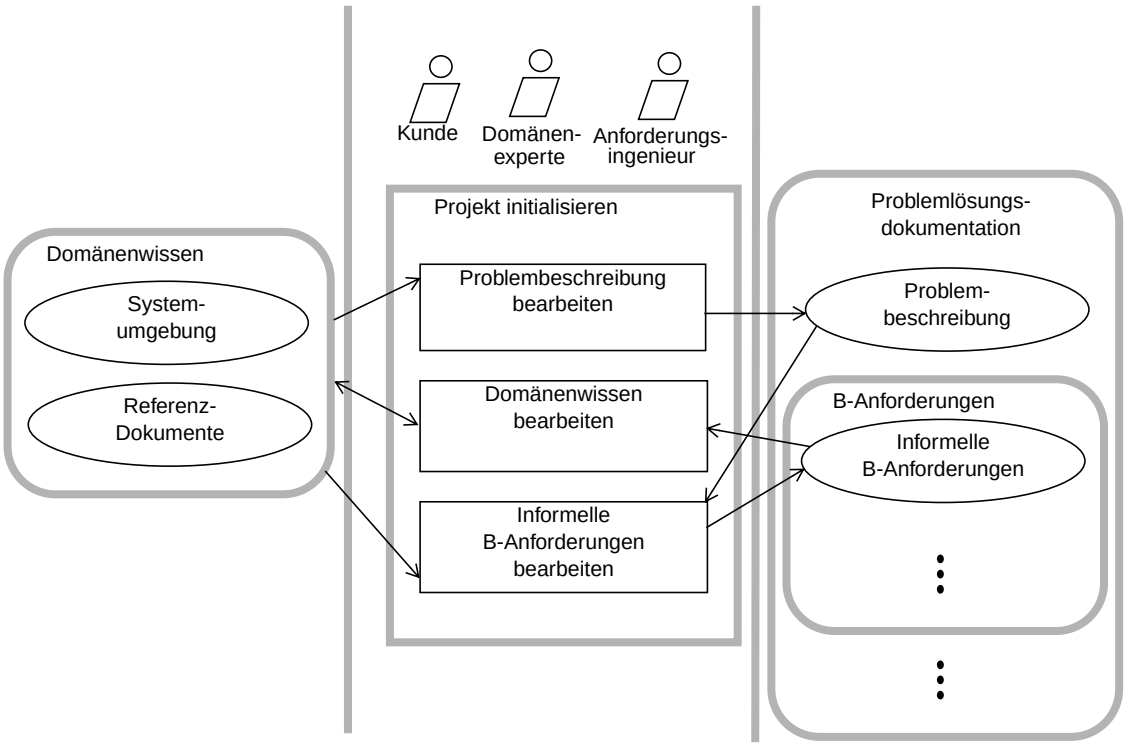


ABBILDUNG 9. Verfeinerung des Prozesses *Projekt initialisieren*

2.2 Produkte

Im Folgenden wird eine Übersicht über die im Prozess *Projekt initialisieren* verwendeten Produkte gegeben. Enthält ein Produkt weitere Produkte, werden alle Teilprodukte aufgeführt, auch wenn sie nicht im Prozess *Projekt initialisieren* benötigt werden (z. B. E-Anforderungen oder A-Verifikation). Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte, soweit dies für das Verständnis des Prozesses *Projekt initialisieren* erforderlich ist.

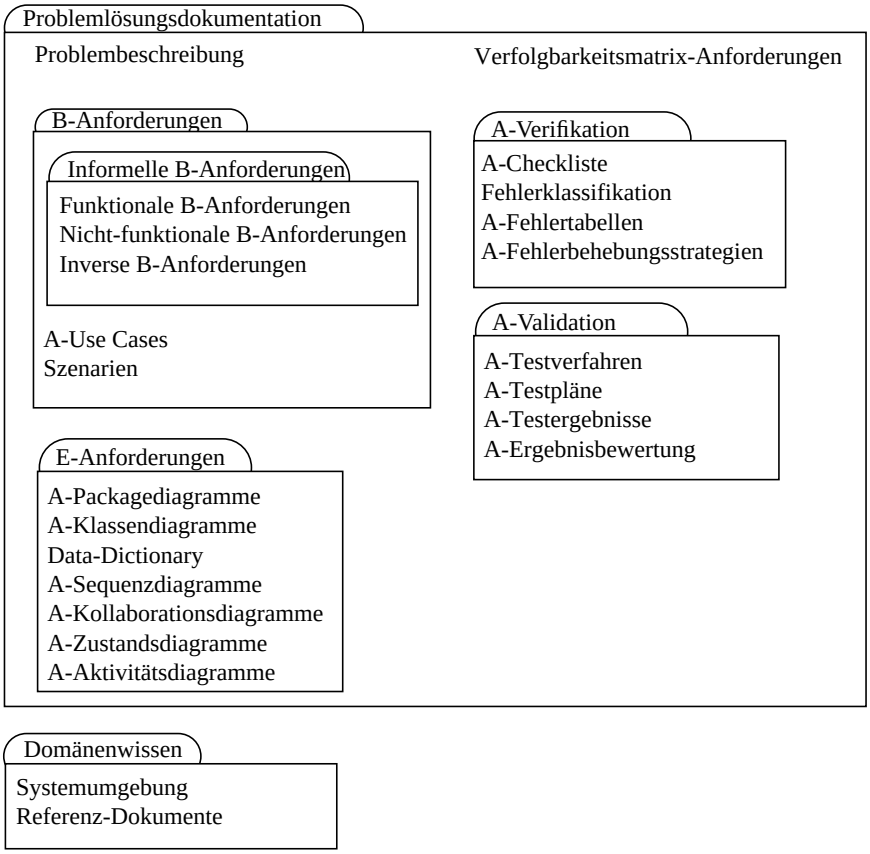


ABBILDUNG 10. Produktübersicht für den Prozess *Projekt initialisieren*

Problembeschreibung

Dieses Dokument beschreibt den Bedarf für ein zu entwickelndes Software-System. Es beschreibt die in einer speziellen Domäne existierenden Probleme oder strategischen Ziele, die durch das Einführen des Systems gelöst oder erreicht werden sollen. Die *Problembeschreibung* enthält außerdem die wichtigsten Anforderungen an die Funktionalität des Systems.

Die *Problembeschreibung* dient als Motivation für die Entwicklung eines eingebetteten Systems. Sie enthält eine erste abstrakte Beschreibung der erwarteten Systemfunktionalität. Sie wird vom Kunden erstellt und dient als Basisdokument für die Entwicklung der informellen B-Anforderungen.

Beispiel *Problembeschreibung*

Problem

Gefordert ist ein Temperatur- und Lichtregelungssystem für ein Universitätsgebäude, da:

- bei der Regelung der Temperatur nicht automatisch auf einen unterschiedlichen Wärmebedarf zu unterschiedlichen Zeiten Rücksicht genommen wird. Die Folge ist, dass Räume auch Nachts bei Nichtbelegung geheizt werden.
- bei der Regelung des Lichts nicht automatisch auf eine Personenanwesenheit geachtet wird. Die Folge ist, dass häufig das Licht in Räumen brennt, obwohl niemand anwesend ist.

Die bisherige Klimaregelung ist wegen überhöhten Energiebedarfs und aufgrund eines Mangels an Komfort nicht zufriedenstellend.

Erwartete Systemfunktionalität

Das zu entwickelnde Kontrollsystem soll die Temperatur- und Lichtregelung für eine „section" innerhalb des Universitätsgebäudes (Gebäude 32, Flur 4 der Universität Kaiserslautern) übernehmen. Dafür verfügt diese „section" über eine Reihe von Installationen (Sensoren und Aktoren). Das Kontrollsystem soll für verschiedene Benutzergruppen ausgelegt sein. Es wird zwischen einem Hausmeister und Benutzern von Zimmern unterschieden. Diese sollen unterschiedliche Funktionalitäten des Systems nutzen können. In jedem Raum soll ein Control Panel bereitgestellt werden, über das der Benutzer seine individuellen Klimaeinstellungen (z. B. Solltemperatur, Arbeitszeiten, Aufheizzeiten; siehe genaue Auflistung weiter unten) eingeben kann. Dem Hausmeister steht ein eigenes Control Panel zur Verfügung über das er verschiedene Werte für das gesamte Gebäude (z. B. Heizperioden, Defaultwerte, Akzeptanzintervalle; siehe genaue Auflistung weiter unten) setzen kann.

ABBILDUNG 11. Ausschnitt aus einer Problembeschreibung

B-Anforderungen

Die *B(enutzer)-Anforderungen* beschreiben die Anforderungen an das zu entwickelnde System aus Sicht des Benutzers/Kunden. Sie gliedern sich in drei Kapitel: 1. die *informellen (tabellarischen) B-Anforderungen*, 2. das Use Case-Diagramm mit einer Beschreibung der einzelnen Use Cases (*A-Use Cases*) und 3. eine Reihe von *Szenarien*, die die *A-Use Cases* verfeinern. Im Folgenden werden nur die für diesen Prozess relevanten *informellen (tabellarischen) B-Anforderungen* erläutert. Eine Beschreibung der *A-Use Cases* und der *Szenarien* erfolgt in Kapitel “System-Anforderungen bearbeiten”.

Informelle B-Anforderungen

Die *informellen B-Anforderungen*[1] beschreiben die *funktionalen*, *nicht-funktionalen* und *inversen B-Anforderungen* an das zu entwickelnde System in tabellarischer Form und in natürlicher Sprache. Die *funktionalen Anforderungen* beschreiben dabei das gewünschte Verhalten des Systems. Die *nicht-funktionalen Anforderungen* beschreiben Qualitätseigenschaften, durch die sich das System auszeichnen soll. Die Qualitätseigenschaften lassen sich teilweise bestimmten Verhalten des Systems zuordnen (z. B. geforderte Reaktionszeiten des Systems bei bestimmten Verwendungen). Teilweise beeinflussen sie aber auch die Struktur des Gesamtsystems (z. B. geforderte Änderbarkeit des Systems). Die *inversen Anforderungen* schließen bestimmte Verhaltensweisen des Systems aus, d. h. sie beschreiben Verhaltensweisen, die das System unter keinen Umständen zeigen darf. Die Einhaltung inverser Anforderungen muss ausführlich getestet werden.

Um die *informellen (tabellarischen) B-Anforderungen* in späteren Entwicklungsdokumenten referenzieren zu können, werden sie durchnummeriert.

Eine tabellarische Auflistung der Anforderungen wird vom Benutzer/Kunden zusammen mit dem Anforderungsingenieur erarbeitet. Abbildung 12 zeigt einen Ausschnitt aus den *informellen B-Anforderungen* für ein Gebäudeautomationssystem.

1. Der Produktname “*Informelle B-Anforderungen*” bezieht sich ausschließlich auf die hier beschriebene tabellarische Darstellung und wird auch im Folgenden so verwendet. Zur Verdeutlichung wird manchmal auch von “*Informellen (tabellarischen) B-Anforderungen”* gesprochen.

Beispiel *Informelle B-Anforderungen*

Die *funktionalen* (BA-F) und *inversen* (BA-Inv) *informellen B-Anforderungen* wurden im Beispiel nach Teilsystemen geordnet. Der Ausschnitt zeigt *funktionale B-Anforderungen* für die Lichtregelung in einem Flurbereich (abgekürzt LichtF) und *inverse* Anforderungen für die Temperaturregelung (abgekürzt Temp). Die *nichtfunktionalen* (BA-NF) *informellen B-Anforderungen* betreffen Qualitätsanforderungen an das Gesamtsystems und lassen sich nicht Teilsystemen zuordnen.

BA-F	Abkürzung	Beschreibung der Anforderung
LichtF-BA-F1	Ausreichend Flurlicht	Wenn ein Flurbereich von einer Person belegt ist, dann soll das Licht in diesem Bereich angeschaltet werden, damit sich die Person sicher bewegen kann (d. h., die Lichtintensität muss ≥ 14 LUX sein).
LichtF-BA-F2	Eintrittslicht	Bevor eine Person einen Flurbereich betritt, muss das Licht in dem zu betretenden Bereich angeschaltet werden, wenn das Licht in diesem Bereich nicht ausreichend ist.

BA-NF	Abkürzung	Beschreibung der Anforderung
BA-NF1	Flexibilität	Das System soll leicht änderbar sein.

BA-Inv	Abkürzung	Beschreibung der Anforderung
Temp-BA-Inv1	Einfrierschutz	Die Raumtemperatur darf nicht unter eine Temperatur fallen, bei der die Heizungsrohre einfrieren. Das System muss garantieren, dass dies niemals passiert.

ABBILDUNG 12. Ausschnitt aus den informellen B-Anforderungen für ein Gebäudeautomationssystem

Domänenwissen

Dieses Dokument beschreibt typische Eigenschaften und Charakteristika der Anwendungsdomäne. Es umfasst insbesondere eine Beschreibung der *Systemumgebung*. Darüber hinaus kann es *Referenzdokumente* umfassen, die z. B. Präzisierungen von Angaben der Problembeschreibung beinhalten.

Systemumgebung

Im Dokument *Systemumgebung* werden spezielle Eigenschaften der Domäne beschrieben. Dies sind z. B. bei Gebäudeautomationssystemen die Architektur des Gebäudes, für das das Software-System entwickelt werden soll, oder Regelungsalgorithmen, mit denen die Temperatur in den Räumen geregelt werden kann. Darüber hinaus sollte die *Systemumgebung* ein Data-Dictionary für die Anwendung enthalten, in der die Begriffe der Anwendung erklärt und beschrieben werden. Das Data-Dictionary (auch: Glossary) ist sehr wichtig, weil es eine gemeinsame Begriffsbildung zwischen den Benutzern/Kunden und den Entwicklern unterstützt. Meist beinhaltet das Dokument *Systemumgebung* außerdem eine Beschreibung der verfügbaren Sensoren und Aktuatoren, die zur Erbringung der Systemfunktionalität bereitstehen. Abbildung 13 zeigt einen Ausschnitt aus einer Gebäudearchitekturbeschreibung.

Beispiel *Gebäudearchitektur*

The total system is composed of a section within an university building (namely section building 32, 4th floor of Univ. of Kaiserslautern) with installation (including sensors and actuators), users, one facility manager and the control system. The control system is composed of hardware and software components. The building is influenced by its environment, mainly the weather.

Building Structure

Sections

The fourth floor of building 32 consists of three sections and shares two staircases SCE and SCW with other floors of the building, as shown in figure 1. A section consists of adjacent or separated groups of rooms, accessible via hallways and/or connecting doors. Sections are divided into offices (O), computer labs (CL), peripheral rooms (P), meeting rooms (M) and hallways (H).

There are three hallways and 22 rooms to control. Figure 1 also shows the six outdoor light sensors (ols1 - ols6) and the major compass directions. The sensors cover the six directions of the different walls. The number in the rooms express the kind and a unique number.

ABBILDUNG 13. Ausschnitt aus der Beschreibung der Gebäudearchitektur

Referenz-Dokumente

In einzelnen Projekten kann es zusätzliche Dokumente geben, die z. B. einzelne Systemaspekte präzisieren oder Erfahrungen aus ähnlichen Projekten schildern. Abbildung 14 und Abbildung 15 zeigen Ausschnitte aus derartigen *Referenz-Dokumenten* (*Referenz-Problembeschreibung* und *Zeit-Referenzmodell*).

Beispiel *Referenz-Problembeschreibung*

Die *Referenz-Problembeschreibung* wurde vom Kunden bereitgestellt und dient dem näheren Verständnis des zu entwickelnden Systems bei Unklarheiten. Die Beschreibung wurde im Rahmen eines vergleichbaren Projekts erstellt.

Description Class LightSensor

INTENTION

This class provides all definitions of a sensor determining whether a light is on or not. Note that this is not a sensor measuring the illuminance. Therefore, on means that the circuit of the light is closed.

BASE CLASSES: Binary Sensor

Intention: All definitions of a binary sensor are also provided here.

DOMAIN KNOWLEDGE

Formal:

$$\Box(noMalSensreactionTime) = 10$$

NL: The reaction time of a light sensor is 10 m

ABBILDUNG 14. Ausschnitt aus der Referenz-Problembeschreibung

Beispiel *Zeit-Referenzmodell*

Das *Zeit-Referenzmodell* fasst Erfahrungen aus einem vorangegangenen Projekt hinsichtlich der Realisierung von Zeitspannen zusammen.

Jahreszeitintervalle

Jahreszeitintervalle ***J*** werden mit [$datum_1$, $datum_2$] angegeben. $Datum_i$ enthält keine Jahresangabe, nur Angaben über Tag und Monat (z.B. 15.09., welches den 15. September eines beliebigen Jahres beschreibt). Dies bedeutet, daß die definierten Intervalle entweder fortlaufend ihre Gültigkeit behalten (Typ ohne Verfallsdatum) oder ungültig werden, sobald sie einmal durchlaufen wurden ("einmalig gültiger Typ"). Ein einmalig gültiger Typ verhält sich bspw. wie der 365-Tage-Timer in einem Videorecorder. Die jeweiligen Intervalle werden um 0:00 Uhr des Tages $datum_1$ gültig und verlieren ihre Gültigkeit um 24:00 des Tages $datum_2$. Das kleinste Intervall ist somit für genau 24h gültig, es wird angegeben mit [$datum_1$, $datum_2$], wobei $datum_1 = datum_2$ ist.

Gilt für ein Intervall $\boldsymbol{J} = [datum_1, datum_2]$

$$datum_2 < datum_1,$$

so gilt: $\boldsymbol{J} = \boldsymbol{J_1} \cup \boldsymbol{J_2}$ mit $\boldsymbol{J_1} = [datum_1, 31.12.]$ und $\boldsymbol{J_2} = [1.1., datum_2]$.

ABBILDUNG 15. Ausschnitt aus dem Zeitreferenz-Modell

2.3 Rollen

Im Prozess *Projekt initialisieren* sind die Rollen *Kunde, Domänenexperte* und *Anforderungsingenieur* involviert. Ihre Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 16. Folgende wesentliche Tätigkeiten sind im Prozess *Projekt initialisieren* mit den Rollen verbunden:

- ***Kunde.*** Der Kunde ist in Zusammenarbeit mit dem Anforderungsingenieur für die Erstellung der *Problembeschreibung* und der *Informellen B-Anforderungen* verantwortlich. Diese Dokumente können weitgehend in natürlicher Sprache formuliert werden, wobei es bezüglich der *Problembeschreibung* keinerlei formelle Vorgaben gibt und für die *Informellen B-Anforderungen* ein tabellarisches Schema vorgegeben wird.
- ***Domänenexperte.*** Die Aufgaben des Domänenexperten sind, das Produkt *Domänenwissen* bereitzustellen und zu warten sowie für Auskünfte über die Domäne bereitzustehen.

- ***Anforderungsingenieur***. Der Anforderungsingenieur unterstützt den Kunden bei der Erstellung der informellen B-Anforderungen. Er hilft, das vorgegebene tabellarische Schema auszufüllen.

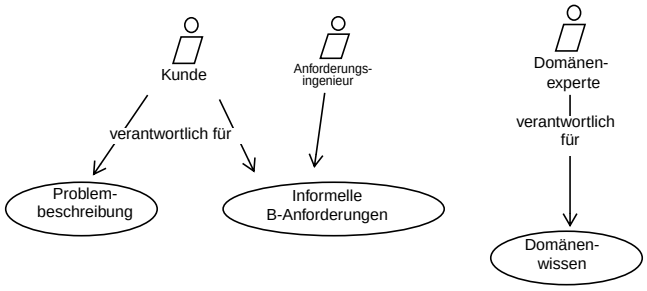


ABBILDUNG 16. Ressourcen und Produkte des Prozesses *Projekt initialisieren*

2.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 17 dargestellt. Der Prozess *Domänenwissen bearbeiten* wird parallel mit den Prozessen *Problembeschreibung bearbeiten* und *Informelle B-Anforderungen bearbeiten* durchgeführt. Eine initiale Version des Produkts Domänenwissen sollte vorhanden sein, bevor die anderen Prozesse bearbeitet werden.

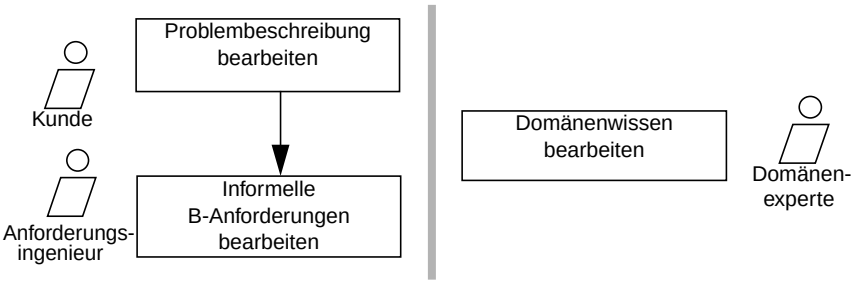


ABBILDUNG 17. Kontrollfluss für den Prozess *Projekt initialisieren*

2.5 Prozess “Problembeschreibung bearbeiten”

Im Prozess *Problembeschreibung bearbeiten* wird der Bedarf für ein eingebettetes System in einem bestimmten Anwendungsbereich identifiziert und dokumentiert. Der Bedarf kann auf verschiedene Arten identifiziert werden: Beispielsweise beobachtet der Kunde seine Umgebung und deckt Probleme auf (z. B. die Regelung der Temperatur erfolgt in den Räumen eines Gebäudes manuell, dies führt zu überhöhtem Energiebedarf) oder der Bedarf ergibt sich aus strategischen Zielen des Unternehmens. Basierend auf dem identifizierten Bedarf werden die wichtigsten Anforderungen an das zu entwickelnde eingebettete System gestellt (z. B. durch eine automatische Regelung der Temperatur soll der überhöhte Energiebedarf gesenkt werden).

Die identifizierten Probleme oder strategischen Ziele und die wichtigsten Anforderungen an ein System sind in der Problembeschreibung dokumentiert.

Vom Kunden hängt ab, wie detailliert die erwartete Sytemfunktionalität an dieser Stelle beschrieben wird.

Folgendes Produkt wird im Prozess *Problembeschreibung bearbeiten* konsumiert:

- Domänenwissen.

Folgendes Produkt wird produziert:

- Problembeschreibung.

2.6 Prozess “Domänenwissen bearbeiten”

Im Prozess *Domänenwissen bearbeiten* werden Eigenschaften und Charakteristika der Anwendungsdomäne vom Domänenexperten dokumentiert. Zur Beschreibung der Domäne gehört eine Dokumentation der späteren Umgebung des zu entwickelnden Software-Systems. In der Beschreibung sollten folgende Fragen geklärt sein:

- Wie sieht die Struktur der Domäne des Systems aus?

  Die Strukturierung des Anwendungsbereichs gibt Aufschluss darüber, wie das spätere System konstruiert werden muss und welche Funktionalität abgedeckt werden soll. Insbesondere sollte beschrieben werden, welche Elemente es im

Anwendungsbereich gibt, welche Beziehungen zwischen den Elementen bestehen und welche Eigenschaften die Elemente haben. Bei der Entwicklung eines Gebäudeautomationssystem sollten z. B. die Räume des Gebäudes, die Anzahl der Fenster und Türen in den verschiedenen Räumen und die Information, ob ein Raum geregelt werden soll, beschrieben werden.

- Welche physikalischen Gesetzmäßigkeiten gelten in der Domäne und welche Beschränkungen ergeben sich daraus?

  Die physikalischen Gesetzmäßigkeiten werden bei der Entwicklung genutzt, um das vom System gewünschte Verhalten zu realisieren. Im Anwendungsbereich Gebäudeautomation gibt es z. B. Gesetzmäßigkeiten über den Wärmeverlust in einem Raum über Fenster und Wände. Teilweise basieren auf den physikalischen Gesetzmäßigkeiten Beschränkungen, die bei der Entwicklung des Systems berücksichtigt werden müssen. Die Berücksichtigung bei der Aufstellung der Anforderungen verhindert, dass Anforderungen gestellt werden, die in der Systemumgebung unmöglich eingehalten werden können. Für ein Gebäudeautomationssystems sind solche Beschränkungen z. B. die maximale Raumtemperatur, die erreicht werden kann, oder die minimale Aufheizzeit von Ist- auf Solltemperatur.

- Welche Sensoren und Aktuatoren sollen im Anwendungsbereich installiert werden?

  Sensoren messen für das System Größen aus der Umgebung und über Aktuatoren beeinflusst das System die Umgebung. Die Beschreibung der Sensoren und Aktuatoren erfolgt gleichzeitig mit der groben Beschreibung der Funktionalität des Systems. Erst wenn klar ist, welche Funktionalität das System bieten soll, kann definiert werden, welche Aktuatoren und Sensoren vom System zur Erbringung der Funktionalität benötigt werden.

Im Dokument “Systemumgebung” sollte es außerdem ein ausgezeichnetes Data-Dictionary geben, in dem die Begriffe des Anwendungsbereichs erläutert werden. Auf diese Weise wird eine Interaktion zwischen dem Kunden und den Entwicklern erleichtert.

Darüber hinaus wird im Prozess *Domänenwissen bearbeiten* geprüft, ob bereits ähnliche Projekte initialisiert wurden. Wenn dies der Fall ist, werden Erfahrungen aus diesen Projekten in Referenz-Dokumenten für die weitere Entwicklung zur Verfügung gestellt.

Folgendes Produkt wird im Prozess *Domänenwissen bearbeiten* konsumiert:

- Informelle B-Anforderungen.

Folgendes Produkt wird modifiziert:

- Domänenwissen.

2.7 Prozess “Informelle B-Anforderungen bearbeiten”

Im Prozess *Informelle B-Anforderungen bearbeiten* werden die in der Problembeschreibung festgehaltenen Anforderungen an das zu entwickelnde System präzisiert und erweitert. Die informellen B-Anforderungen werden durch den Kunden in Zusammenarbeit mit dem Anforderungsingenieur erstellt.

Es sollte zwischen funktionalen, nicht-funktionalen und inversen Anforderungen unterschieden werden. Die funktionalen Anforderungen verfeinern die in der Problembeschreibung festgehaltene erwartete Systemfunktionalität. Sie lassen sich beispielsweise nach verschiedenen Systemfunktionen strukturieren (z. B. Temperatur- und Lichtregelung). Es gilt abzugleichen, ob die beschriebene Systemfunktionalität mit den zur Verfügung stehenden Sensoren und Aktuatoren erreicht werden kann.

Evtl. hat der Kunde nicht-funktionale Anforderungen an das zu entwickelnde System. Beispielsweise wünscht er sich bestimmte Reaktionszeiten des Systems. Die Dokumentation der nicht-funktionalen Anforderungen kann informell geschehen. Später sollte darauf geachtet werden, dass alle nicht-funktionalen Anforderungen quantifizierbar dokumentiert sind. Auf diese Weise kann ihre Einhaltung überprüft werden.

Sollen bestimmte Verhaltensweisen des Systems ausgeschlossen werden, dann können diese Anforderungen als inverse Anforderungen festgehalten werden.

Folgende Produkte werden im Prozess *Informelle B-Anforderungen bearbeiten* konsumiert:

- Domänenwissen.
- Problembeschreibung.

Folgendes Produkt wird produziert:

- Informelle B-Anforderungen.

2.8 Weiterführende Literatur

1. Merlin Dorfman, Richard H. Thayler: *Standards, Guidelines and Examples on System and Software Requirements Engineering.* IEEE Computer Society Press, 1990.
2. IEEE Std 1076-1993. IEEE, 1994.
3. Richard H. Thayer, Merlin Dorfman: *Software Requirements Engineering*. Second Edition, Institute of Electrical & Electronic Engineering, 1999.

KAPITEL 3

System-Anforderungen bearbeiten

Ziel des Prozesses *System-Anforderungen bearbeiten* ist die Ermittlung und umfassende Beschreibung von Anforderungen und Beschränkungen, denen das zu entwickelnde eingebettete System genügen soll. Diese werden in der *Problemlösungsdokumentation* fixiert. Im Laufe des Prozesses sollen die Benutzer/Kunden und die Entwickler ein gemeinsames und möglichst genaues Verständnis des gewünschten Systems bekommen. Dafür müssen sich die Entwickler in das Anwendungsgebiet und die Benutzer in Möglichkeiten der Informationstechnik einarbeiten.

Abbildung 18 gibt einen Überblick über die verwendeten Produkte im Prozess *System-Anforderungen bearbeiten*.

Folgende Produkte werden konsumiert:

- *Domänenwissen*. Dieses Dokument beschreibt typische Eigenschaften und Charakteristika der Anwendungsdomäne. Es umfasst insbesondere eine Beschreibung der Systemumgebung.
- *Namenskonventionen*. Dieses Dokument beinhaltet Konventionen für die Wahl von Bezeichnern in der *Problemlösungsdokumentation*.

Folgendes Produkt wird im Prozess *System-Anforderungen bearbeiten* modifiziert:

- *Problemlösungsdokumentation.* Dieses Dokument umfasst neben einer Problembeschreibung eine Beschreibung der Anforderungen aus Sicht des Benutzers/Kunden und eine Beschreibung der Anforderungen aus Sicht der Entwickler. Außerdem umfasst das Dokument eine Verfolgbarkeitsmatrix, mit deren Hilfe Zusammenhänge innerhalb der System-Anforderungen explizit gemacht werden. Des Weiteren enthält die *Problemlösungsdokumentation* Dokumente zu Verifikationsaktivitäten auf System-Anforderungsebene (z. B.: eine Fehlerklassifikation, eine Checkliste zur Inspektion der System-Anforderungen, Fehlertabellen zur Dokumentation gefundener Fehler und Fehlerursachen) und Dokumente zu Validationsaktivitäten (z. B. Testpläne, Fehlertabellen zur Dokumentation gefundener Fehlverhalten[1], Fehler[2] und Fehlerursachen[3]).

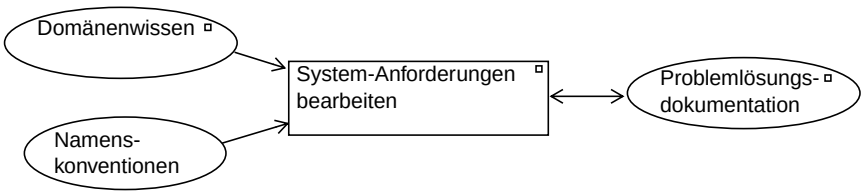


ABBILDUNG 18. Verwendete Produkte des Prozesses *System-Anforderungen bearbeiten*

3.1 Prozessverfeinerung

Der Prozess *System-Anforderungen bearbeiten* unterteilt sich in fünf verschiedene Teilprozesse (siehe Abbildung 19). Im Prozess *B-Anforderungen bearbeiten* werden die Anforderungen aus Benutzersicht (die sog. *B(enutzer)-Anforderungen*) vervollständigt. Dafür ist ein Verständnis der Domäne (*Domänenwissen*) wichtig. Die

1. Fehlverhalten (Failure): Abweichung zwischen dem tatsächlichen Verhalten des ausgeführten Systems und seinem geforderten/erwarteten Verhalten.
2. Fehler (Fault): In der Dokumentation (d. h., in einem Produkt) befindliches Problem (z. B. Schnittstellenfehler, Initialisierungsfehler oder unklare Beschreibung). Ein Fehler kann zu Fehlverhalten des Systems führen.
3. Fehlerursache (Error): Menschlicher Defekt/Fehlhandlung, der zu Fehlern führt (z. B. Nichtverstehen des zu lösenden Problems).

B-Anforderungen werden im Prozess *E-Anforderungen bearbeiten* in eine für Entwickler vertraute Notation überführt und verfeinert (die sog. *E(ntwickler)-Anforderungen*). Beim Verfeinerungsprozess fließt erneut Wissen aus der Domäne ein. *Namenskonventionen* gewährleisten eine einheitliche Namengebung in allen weiteren Entwicklungsprodukten. Zusätzlich wird im Prozess *Verfolgbarkeitsmatrix-Anforderungen bearbeiten* eine Verfolgbarkeitsmatrix aufgestellt (die sog. *Verfolgbarkeitsmatrix-Anforderungen*), in der Zusammenhänge zwischen den *B-Anforderungen* und den *E-Anforderungen* beschrieben werden. Im Prozess *System-Anforderungen verifizieren* werden die *B-Anforderungen* gegen die *Problembeschreibung* und die *E-Anforderungen* gegen die *B-Anforderungen*, sowie die *Verfolgbarkeitsmatrix-Anforderungen* verifiziert. Die Ergebnisse der Verifikation werden im Dokument *A-Verifikation* dokumentiert und gefundene Fehler werden behoben. Im Prozess *Testfälle bearbeiten* werden aus den System-Anforderungen Testpläne für spätere Tests fertiggestellter Teilsysteme und des Gesamtsystems abgeleitet. Die erarbeiteten Testpläne werden im Dokument *A-Validation* dokumentiert.

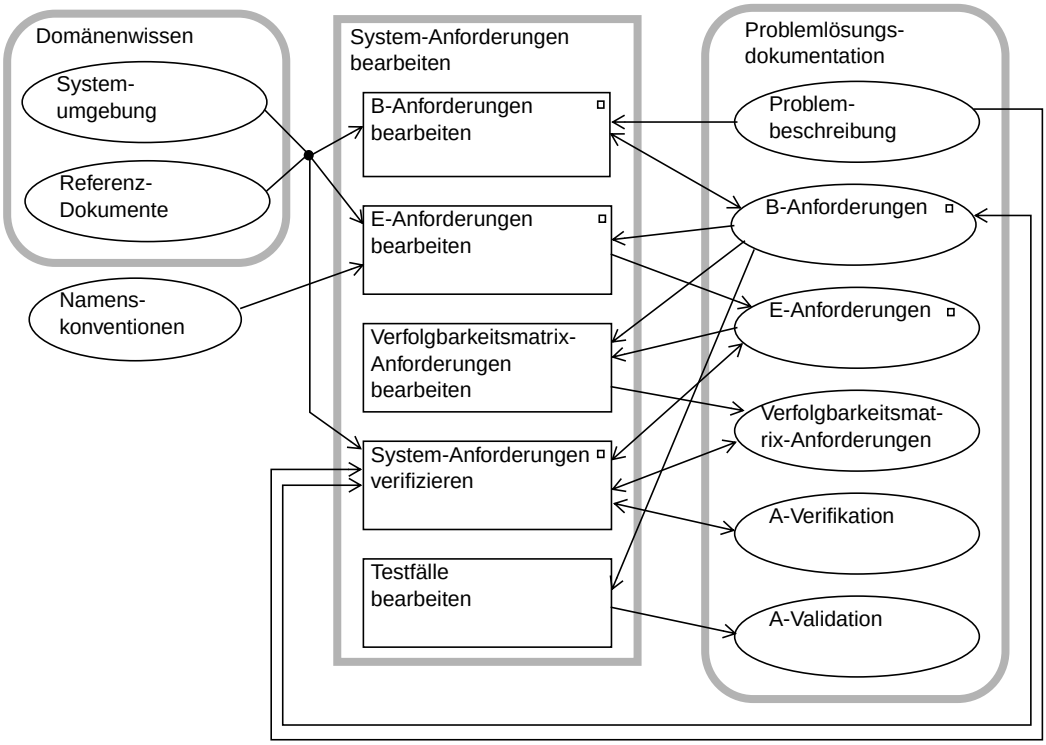


ABBILDUNG 19. Verfeinerung des Prozesses *System-Anforderungen bearbeiten*

3.2 Produkte

Im folgenden wird eine Übersicht über die im Prozess *System-Anforderungen bearbeiten* verwendeten Produkte gegeben. Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte der Dokumente, soweit dies nicht bereits im vorangegangenen Kapitel erfolgt ist.

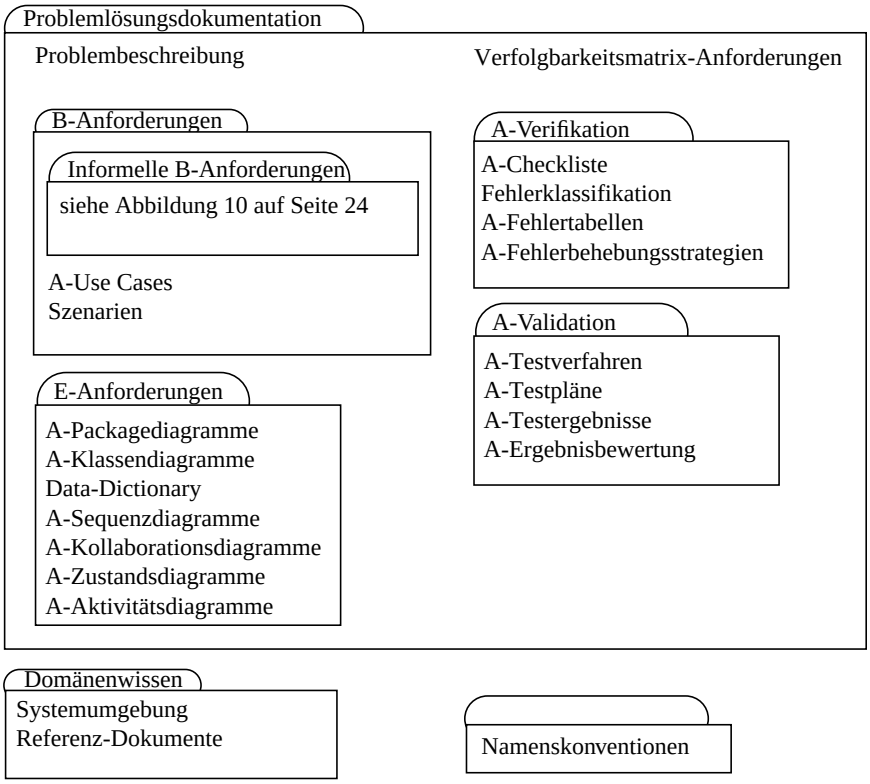


ABBILDUNG 20. Produktübersicht für den Prozess *System-Anforderungen bearbeiten*

B-Anforderungen

Die *B-Anforderungen* beschreiben die Anforderungen an das zu entwickelnde System aus Sicht des Benutzers/Kunden. Sie gliedern sich in drei Kapitel: 1. die *informellen (tabellarischen) B-Anforderungen*, 2. das Use Case-Diagramm[1] mit einer

1. Da Use Cases auch auf Entwurfsebene auftreten können, werden die Use Cases auf Anforderungsebene im Folgenden auch als *A-Use Cases* bezeichnet. Dies gilt analog auch für weitere Bezeichner mit dem Präfix “A-”.

kurzen Beschreibung der einzelnen Use Cases (*A-Use Cases*) und 3. eine Reihe von *Szenarien*, die die *A-Use Cases* verfeinern. Die *informellen (tabellarischen) B-Anforderungen* wurden bereits im Kapitel “Projekt initialisieren” erläutert, daher werden im Folgenden nur die *A-Use Cases* und die *Szenarien* beschrieben.

Use Case-Diagramm (A-Use Cases)

Das *Use Case-Diagramm* umfasst die möglichen *Use Cases* (d. h., die Anwendungsfälle) für das zu entwickelnde System. Damit stellt es die möglichen Verwendungsweisen des Systems dar. Es überdeckt die funktionalen Anforderungen an das System. Ein *Use Case* beschreibt eine Aktivität, die durch das zu entwickelnde System unterstützt werden soll. Dies kann eine konkrete Systemfunktionalität sein (z. B. das Aufdrehen des Heizkörperventils zum Heizen). Die Aktivität kann aber auch eine abstrakte Beschreibung vom gewünschten Verhalten des Systems sein (z. B. die Raumtemperatur soll auf die gewünschte Temperatur steigen). Ein *Use Case* umfasst eine Beschreibung der Berührungspunkte der Aktivität zu ihrer Umwelt (z. B. welche Benutzer sind an der Aktivität beteiligt). Ein *Use Case* wird durch einen Akteur ausgelöst. In eingebetteten Systemen können Akteure von *Use Cases* einerseits außerhalb des zu entwickelnden Systems liegen, wie z. B. ein Benutzer. Andererseits können Akteure auch Teil des Systems sein, wie z. B. Sensoren. Zum Teil beeinflussen externe Akteure interne Akteure derart, dass eine Systemreaktion ausgelöst wird, beispielsweise wenn ein Benutzer den Flur betritt und der Anwesenheitssensor Anwesenheit meldet. Der *Use Case* endet mit einem nach außen sichtbaren Verhalten.

Die vom Use Case beeinflussten Größen in der Umgebung werden im Use Case-Diagramm ebenfalls als Akteure dargestellt. In eingebetteten Systemen führt ein Systemverhalten häufig zu einem Ansteuern von Aktuatoren, z. B. einer Lampe. Die Aktuatoren sind Teil des Systems und werden damit als interne Akteure modelliert. Die beeinflusste Größe der Umgebung (repräsentiert als externer Akteur) ist beispielsweise die Helligkeit im Raum.

Das Use Case-Diagramm kann helfen, Sensoren und Aktuatoren, die für die Verwendungsweisen des Systems benötigt werden, zu bestimmen. Im Folgenden wird von dem einfachen Fall ausgegangen, dass die notwendigen Sensoren und Aktuatoren bereits bekannt und im Dokument Domänenwissen beschrieben sind.

Es gilt zu überprüfen, ob alle vorgesehenen Aktuatoren (siehe Dokument Domänenwissen) auch tatsächlich für das Verhalten des Systems benötigt werden und umgekehrt. Abbildung 21 zeigt einen Ausschnitt aus einem *Use Case-Diagramm*

mit einer Erläuterung eines *Use Cases*. Die Beschreibung eines Use Cases geschieht nach einem vorgegebenen Schema:

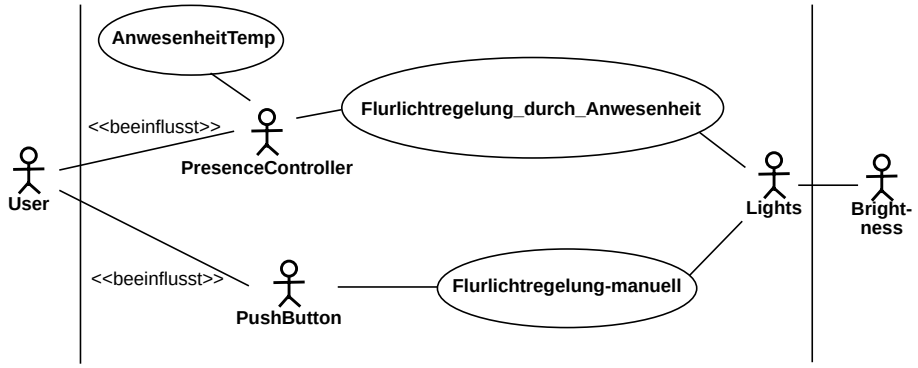
- **Initiating event** beschreibt, durch welche Aktion eines “actors” der Use Case ausgelöst wird.
- **Pre condition** beschreibt den Ausgangszustand, der gegeben ist, bevor der Use Case startet.
- **Description** beschreibt, welche Schritte im Use Case durchgeführt werden.
- **Expected behavior** beschreibt, was das Ziel des Use Cases ist und womit der Use Case endet.
- **Exceptions** beschreiben mögliche Ausnahmen und Abweichungen vom Pfad, der in der “Description” beschrieben wurde.
- **Restrictions** beschreiben nicht-funktionale Anforderungen, die sich einzelnen Verwendungen des Systems zuordnen lassen.
- **Post condition** beschreibt den Zustand nach Durchführung des Use Cases.

Das *Use Case-Diagramm* mit Erläuterungen zu den einzelnen *Use Cases* dient als Diskussionsgrundlage zwischen Benutzern/Kunden und Entwicklern, um ein gemeinsames Verständnis des gewünschten Systems und des Anwendungsgebiets zu fördern. Jeder abstrakte *Use Case* wird mit einer Reihe von *Szenarien* veranschaulicht.

Beispiel *Use Case-Diagramm (A-Use Cases)*

Das Beispiel zeigt den *Use Case* “Flurlichtregelung_durch_Anwesenheit”. Jeder *Use Case* wird mit Hilfe eines vorgegebenen Schemas erläutert. Außerdem werden zu jedem *Use Case* die zugehörigen *Szenarien* angegeben. Der *Use Case* “Flurlichtregelung_durch_Anwesenheit” wird durch einen Benutzer (engl.: User; externer Akteur) ausgelöst, der einen Flurabschnitt betritt oder verlässt. Die Personenan- oder -abwesenheit wird durch den PresenceController (interner Akteur), der Teil des Systems ist, festgestellt. Der Anwesenheitssensor wurde in diesem Beispiel nicht als Akteur modelliert, da er ein passiver Sensor ist und damit die Systemreaktion nicht direkt auslösen kann. Die aktive Komponente PresenceController fragt den Sensor laufend ab und löst die Systemreaktion aus. Der *Use Case* führt dazu, dass das Licht (angesteuerter Aktuator) in einem Flurabschnitt an- oder abgeschaltet wird. Das Licht (interner Akteur) beeinflußt die Helligkeit (externer Akteur) als beobachtete Größe in der Umgebung.

Das Use Case-Diagramm wurde mit Hilfe des CASE-Werkzeugs StP/UML erstellt. Ein Rechteck innerhalb eines *Use-Cases* deutet an, dass eine weitere Verfeinerung des *Use Cases* in Form von einer Menge von A-Sequenzdiagrammen existiert.



Use Case: Flurlichtregelung_durch_Anwesenheit

Initiating event: Eine Person betritt oder verläßt den Flur. Der “PresenceController” meldet die An- bzw. Abwesenheit von Personen dem System.

Pre condition: Der Flur ist seit ZeitspanneVerlassen-Minuten nicht belegt und die Helligkeit im Flur ist <= 14 Lux oder eine Person ist im Raum anwesend und die Lampe ist angeschaltet.

Description: Wenn die Anwesenheit von Personen im Flur registiert wird, wird das Licht eingeschaltet. Wenn der Flur seit ZeitspanneVerlassen-Minuten nicht belegt ist, wird das Licht ausgeschaltet.

Expected behavior: Die Lampe wird an- oder ausgeschaltet.

Postcondition: Eine Person ist im Flurabschnitt anwesend und die Lampe ist angeschaltet oder keine Person ist seit ZeitspanneVerlassen-Minuten im Flurabschnitt und die Lampe ist ausgeschaltet.

Restrictions: ----

Combination with Use Case: ----

Dieser Use Case wird in folgenden Szenarien verdeutlicht:

- Flurlichtregelung_durch_Anwesenheit_Szenario1
- Flurlichtregelung_durch_Anwesenheit_Szenario2
- Flurlichtregelung_durch_Anwesenheit_Szenario3

ABBILDUNG 21. Ausschnitt aus einem Use Case-Diagramm mit Erläuterung des Use Cases “Flurlichtregelung_durch_Anwesenheit”

Szenarien

Ein *Use Case* repräsentiert ein mögliches Verhalten des Systems abstrakt (z. B. das Licht im Flur wird durch Anwesenheit geregelt). Ein *Szenario* beschreibt eine konkrete Interaktionssequenz zwischen einem Akteur und dem System, die zu einem Systemverhalten führt. Für einen *Use Case* kann es verschiedene *Szenarien* geben, die abhängig von den konkreten Eingabevariablen sind (z. B. eine Person betritt den Flur oder eine Person verlässt den Flur). Für jedes mögliche "initiating event" eines *Use Cases* wird ein Szenario entwickelt.

Die *Szenarien* werden nach den zugehörigen *Use Cases* durchnummeriert. Jedes *Szenario* enthält eine kurze Beschreibung und die Vorbedingungen, die zum Durchlaufen des *Szenarios* erfüllt sein müssen (letztere können z. B. im ersten Schritt des Szenarios beschrieben werden). Für jedes in den *informellen funktionalen* und *inversen B-Anforderungen* beschriebene Verhalten sollte ein *Szenario* existieren. Zu überprüfen, ob dies geschehen ist, kann durch eine Referenzierung der *informellen (tabellarischen) B-Anforderungen* im Szenario erleichtert werden. Ein *Szenario* endet mit einer nach außen hin sichtbaren Reaktion des Systems (z. B. der Ansteuerung eines Aktuators). In einem Szenario kann auch beschrieben werden, dass das System in einer bestimmten Situation nicht reagiert (z. B. wenn eine Person einen Raum vorzeitig wieder verlässt und das System aus diesem Grund nicht regelnd eingreift). *Informelle nicht-funktionale B-Anforderungen*, die sich einzelnen Verwendungen des Systems zuordnen lassen, werden in der Szenario-Beschreibung vermerkt. Abbildung 22 zeigt eine mögliche Beschreibung eines Szenarios.

Beispiel *Szenario*

Das Beispiel zeigt ein *Szenario* zur Regelung des Flurlichts, d. h. der *Use Case* "Flurlichtregelung_durch_Anwesenheit" wird verfeinert. Das *Szenario* beschreibt den Fall, dass ein Benutzer einen leeren Flur betritt.

Nr.	Schritt	Beschreibung der Szenarien	B-Anf
Szenario1	*Szenario1 für Use Case Flurlichtregelung_durch_Anwesenheit* *Ein Benutzer betritt den leeren Flur.*		
	1.	**Keine Anwesenheit** (Anwesensheitsdetektor (imd3) = 0) **Flurlicht aus** (status_line sl1 = “off”)	
	2.	Der Flur ist leer, ein Benutzer betritt den Flur durch die Flurtür.	
	3.	Das Flurlicht wird eingeschaltet. (Annahme: Die erforderliche Mindestlichtstärke wird auch bei geöffneten Türen nie erreicht).	LichtF-BA-F1 LichtF-B-F2

ABBILDUNG 22. Beschreibung des Szenarios1 - Flurlichtregelung durch Anwesenheit

E-Anforderungen

Die *E-Anforderungen* beschreiben das Verhalten des zu entwickelnden Software-Systems aus Sicht der Entwickler. Dafür wird das System in Komponenten (Klassen) aufgeteilt. Das erwartete Verhalten wird basierend auf der Interaktion der Komponenten (auf Objektebene) beschrieben. Es werden sieben verschiedene UML-Diagramme zur Beschreibung des gewünschten Systems verwendet: 1. *Packagediagramme*, 2. *Klassendiagramme*, 3. ein *Data-Dictionary*, 4. *Sequenzdiagramme*, 5. *Kollaborationsdiagramme*, 6. *Zustandsdiagramme* und 7. *Aktivitätsdiagramme*.

A-Packagediagramme

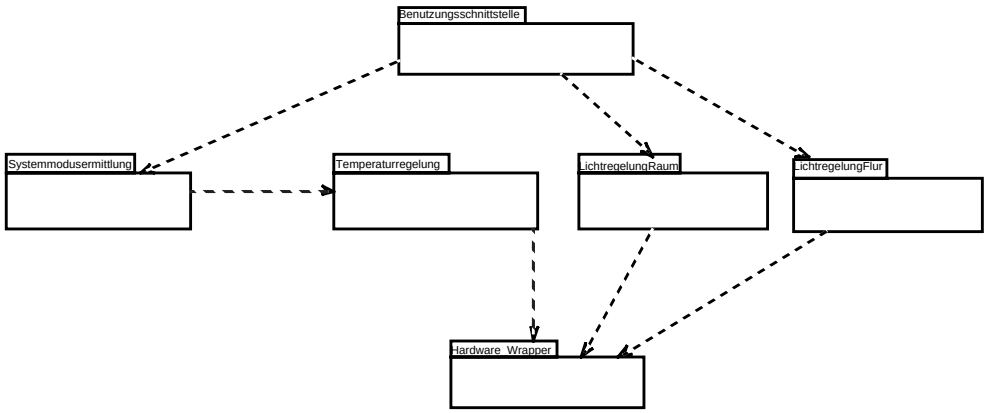
Bei großen Systemen is es sinnvoll, ausgezeichnete *Packagediagramme* zu erstellen, die eine Aufteilung des Gesamtsystems in Teilsysteme aufzeigen und Beziehungen zwischen den existierenden Teilsystemen verdeutlichen (statische Struktur des Systems). Derartige *Packagediagramme* werden wieder durch *Packagediagramme* oder auf unterster Ebene durch Klassendiagramme verfeinert. Eine Aufteilung in Teilsysteme reduziert die Komplexität des Gesamtsystems und erlaubt die parallele Bearbeitung verschiedener Teilsysteme. Auf Anforderungsebene lassen sich bereits erste Teilsysteme anhand der Funktionalitäten, die das System über-

nehmen soll, identifizieren. Im Entwurf werden die bereits identifizierten Teilsysteme weiter verfeinert.

Zum besseren Verständnis sollte jedes *Package* kurz erläutert werden. Abbildung 23 zeigt ein *Packagediagramm* für das Gebäudeautomationssystem mit einer Erläuterung des *Package* “LichtregelungFlur”.

Beispiel *A-Packagediagramm*

Im betrachteten Gebäudeautomationssystem wurde eine Aufteilung in sechs Teilsysteme (Benutzungsschnittstelle, Systemmodusermittlung, Temperaturregelung, LichtregelungRaum, LichtregelungFlur und Hardware Wrapper) vorgenommen, um eine frühzeitige parallele Bearbeitung des Systems zu erlauben. Die Teilsysteme Systemmodusermittlung, Temperaturregelung, LichtregelungRaum und LichtregelungFlur wurden anhand der gewünschten Funktionalitäten des zu entwickelnden Systems gebildet. Sie sind weitgehend unabhängig und können parallel entwickelt werden. Das Teilsystem Benutzungsschnittstelle bildet die Schnittstelle zum Benutzer und greift zum Setzen von verschiedenen Werten auf die funktionalen Teilsysteme zu. Das Teilsystem Hardware Wrapper bildet die Schnittstelle zu den konkreten Sensoren und Aktuatoren. Die funktionalen Teilsysteme verwenden dieses Teilsystem zur Abfrage von Sensorwerten und zum Ansteuern von Aktuatoren.



Package: LichtregelungFlur
Dieses Package enthält Klassen, die die Regelung des Lichts in den Flurabschnitten übernehmen. Werte, die dafür benötigt werden, werden mit Hilfe der Klassen des Packages Benutzungsschnittstelle gesetzt.

ABBILDUNG 23. Teilsysteme eines Gebäudeautomationssystems und Erläuterung des Package "LichtregelungFlur"

A-Klassendiagramme

Ein *Klassendiagramm* beschreibt *Klassen*, Attribute und Methoden von *Klassen* und Beziehungen zwischen *Klassen* innerhalb eines Teilsystems. Es beschreibt damit die statische Struktur eines Teilsystems. Für jedes elementare Teilsystem (das sind Teilsysteme, die keine weiteren Packages enthalten) existiert ein *Klassendiagramm*.

In jedem *Klassendiagramm* sind die Methoden und Attribute einer *Klasse* dargestellt, die für die Erbringung der im *Klassendiagramm* beschriebenen Funktionalität erforderlich sind. Tritt eine *Klasse* in mehreren *Klassendiagrammen* auf (d. h., wird sie zur Erbringung unterschiedlicher Funktionalitäten benötigt), dann zeigen einzelne *Klassendiagramme* nur Sichten auf diese *Klasse*.

Die *Klassen* sollten sich an den Objekten der Anwendungsdomäne orientieren (z. B. an der Gebäudearchitektur bei einem Gebäudeautomationssystem). Außerdem werden für Sensoren und Aktuatoren *Klassen* gebildet. Dies erhöht die Verständlichkeit des Systems. Zum Entwurf hin kann über ein Streichen dieser Klassen aus Gründen der Effizienz des Systems nachgedacht werden. Hauptaufgabe der A-Klassendiagramme ist, das Verständnis des zu entwickelnden Systems

zu erleichtern. Sensoren und Aktuatoren werden durch kontrollierende *Klassen* gekapselt. Bei Sensoren übernimmt die kontrollierende *Klasse* ständige Abfragen. Bei Aktuatoren löst die kontrollierende *Klasse* Konflikte zwischen verschiedenen Funktionalitäten (z. B. die Temperaturregelung will die Jalousien schließen und die Lichtregelung will sie öffnen). Komplexe Regelungsaufgaben lassen sich durch Hierarchien von Regelungsklassen vereinfachen.

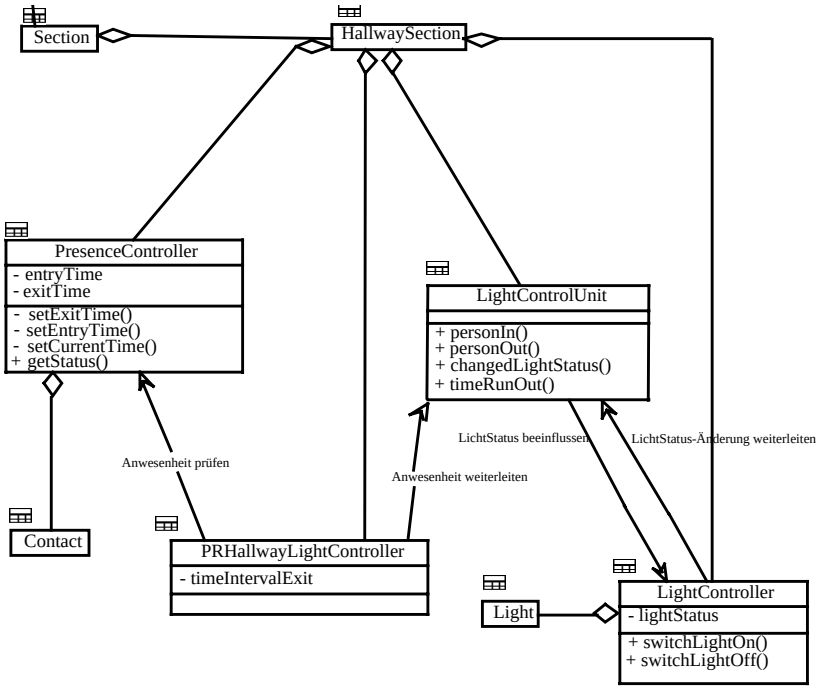
Das Klassendiagramm wird mit einer kurzen Beschreibung versehen.

Abbildung 24 zeigt einen Ausschnitt aus dem *Klassendiagramm* des Teilsystems “LichtregelungFlur” des Gebäudeautomationssystems.

Beispiel *A-Klassendiagramm*

Die *Klassen* spiegeln zum Teil die statische Gebäudearchitektur wieder (z. B. besteht eine Section aus HallwaySections). Für Sensoren und Aktuatoren wurden *Klassen* gebildet (z. B. Contact oder Light). Die Sensoren und Aktuatoren werden durch kontrollierende *Klassen* gekapselt (z. B. PresenceController oder LightController). Außerdem wurde im Beispiel eine Hierarchie von Kontrolleinheiten gebildet. Beispielsweise ist der “PRHallwayLightController” der “LightControlUnit” untergeordnet und der “PresenceController” ist wiederum dem “PRHallwayLightController” untergeordnet.

Das *Klassendiagramm* wurde mit dem CASE-Werkzeug StP/UML erstellt. Ein Minus vor einem Attribut oder einer Methode bedeutet, dass es sich um ein internes Attribut oder eine interne Methode handelt (nach außen nicht sichtbar). Ein Plus kennzeichnet eine öffentliche Methode (öffentliche Attribute sollten aus Gründen des “Information Hidings” vermieden werden). Eine Tabelle an der linken oberen Ecke einer *Klasse* zeigt an, dass zu der *Klasse* eine Klassentabelle existiert. In dieser können Attributen und Methoden zusätzliche Eigenschaften zugewiesen werden.



Klassendiagramm LichtregelungFlur
Dieses Diagramm zeigt alle Klassen, die an der Erbringung der Funktionalität der Lichtregelung in den Flurabschnitten beteiligt sind.

ABBILDUNG 24. Klassendiagramm des Teilsystems LichtregelungFlur

Data-Dictionary

Im *Data-Dictionary* der E-Anforderungen werden alle Klassen und die zu der jeweiligen Klasse gehörigen Attribute, Methoden und Assoziationen beschrieben. Im Gegensatz zu den *Klassendiagrammen* beschreibt das *Data-Dictionary* keine Sichten auf eine Klasse, sondern fasst die Informationen bez. einer Klasse aus allen Klassendiagrammen zusammen. Abbildung 25 zeigt einen Ausschnitt aus einem *Data-Dictionary* für die Klasse “LightControlUnit” aus dem Teilsystem “LichtregelungFlur” des Gebäudeautomationssystems.

Beispiel *Data-Dictionary*

Das Beispiel zeigt einen Ausschnitt der informellen Beschreibung der Klasse “LightControlUnit’ aus dem *Data-Dictionary.* Erläutert wird die Klasse mit Ihren Attributen, Methoden und Beziehungen. Die Methoden fmSwitchOff() und fmSwitchOn() sind im Klassendiagramm der LichtregelungFlur nicht enthalten. Sie sind im Klassendiagramm Benutzungsschnittstelle beschrieben.

Zu beschreibendes Objekt	Attribut, Methode, Assoziation	Beschreibung
LightControlUnit	Diese Klasse beschreibt eine Einheit, die aus den Informationen über Anwesenheit im Flur und über den Status der Lichtanlage das entsprechende Verhalten dieser Lichtanlage auslöst.	
	fmSwitchOff()	Diese Methode wird aufgerufen, falls der Hausmeister das Licht im Flur über das ControlPanelFM ausschalten möchte.
	fmSwitchOn()	Diese Methode wird aufgerufen, falls der Hausmeister über den ControlPanelFM das Licht im Flur einschalten möchte.
	personIn()	Durch diese Methode erfolgt die Benachrichtigung vom PRHallwayLightController, dass jemand in den Flur eingetreten ist.
	personOut()	Durch diese Methode erfolgt die Benachrichtigung vom PRHallwayLightController, dass alle Benutzer aus dem Flurabschnitt ausgetreten sind.
	changedLightStatus()	Diese Methode wird vom LightController aufgerufen, falls der Zustand der Lichtanlage im Flur sich geändert hat, d.h. der Benutzer von Hand das Licht an oder ausgeschaltet hat.
	timeRunOut()	Über diese Methode meldet die Klasse PRHallwayLightController, dass der Flur seit länger als ZeitspanneVerlassen (LT2) Minuten leer ist.

ABBILDUNG 25. Auszug aus der Beschreibung der Klasse LightControlUnit

A-Sequenzdiagramme

Die *A-Sequenzdiagramme* verfeinern die Szenarien aus den B-Anforderungen. Sie bilden das in den Szenarien beschriebene Verhalten des Systems auf die Interaktion zwischen Objekten innerhalb des Systems ab. Die Anzahl der *Sequenzdiagramme* entspricht der Anzahl der Szenarien. Für Szenarien, die beschreiben, dass das Sys-

tem nicht reagiert, müssen keine *Sequenzdiagramme* entwickelt werden. *Sequenzdiagramme* betonen den zeitlichen Ablauf des Nachrichtenaustauschs zwischen Objekten im System. Abbildung 26 zeigt ein *Sequenzdiagramm* zu “Szenario1” des Use Cases “Flurlichtregelung_durch_Anwesenheit”.

Beispiel *A-Sequenzdiagramm*

Der “PresenceController” ist in diesem *Sequenzdiagramm* als auslösender Akteur besonders gekennzeichnet. Er fragt den Anwesenheitssensor regelmäßig ab und löst durch seine Zustandsänderung eine Systemreaktion aus. Der “PRHallway-LightController” ist - wie der PresenceController - ein aktives Objekt, dass den Zustand des “PresenceControllers” regelmäßig abfragt. Die Trennung von PRHallwayLightController und PresenceController wurde vorgenommen, um den PresenceController möglichst allgemein zu halten. Er wird mit gleicher Funktionalität auch für die Temperaturregelung in den Räumen eingesetzt. Wenn jemand den Raum betritt, wird an die “LightControlUnit” “personIn()” geschickt und der “LightController” wird dazu veranlasst, das Licht anzuschalten. Wenn das Licht nicht bereits angeschaltet ist, schaltet der “LightController” es an.

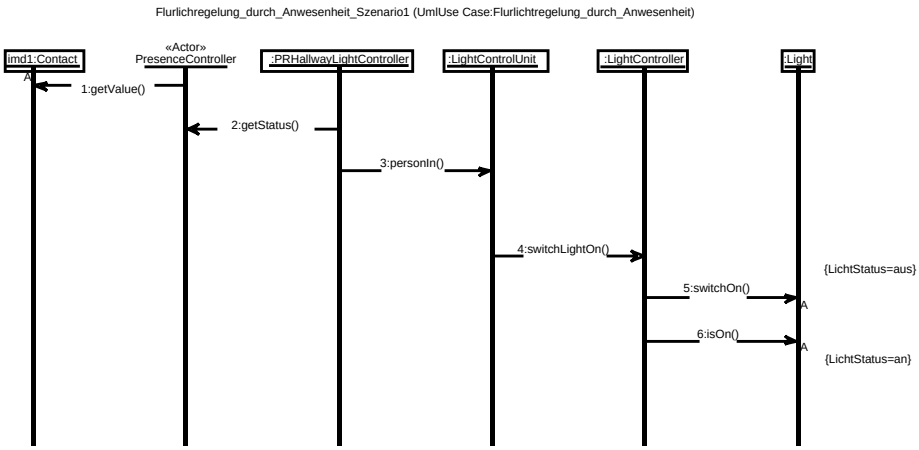


ABBILDUNG 26. Ausschnitt aus dem Sequenzdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit

A-Kollaborationsdiagramme

Die *Kollaborationsdiagramme* verfeinern wie die Sequenzdiagramme die Szenarien aus den B-Anforderungen. Sie bilden das in den Szenarien beschriebene Verhalten des Systems auf die Interaktion zwischen Objekten innerhalb des Systems ab. Die *Kollaborationsdiagramme* betonen die Zusammenarbeit von Objekten.

Mit Hilfe eines CASE-Werkzeugs (z. B. des CASE-Werkzeugs StP/UML) lassen sich die *Kollaborationsdiagramme* automatisch aus den Sequenzdiagrammen generieren. Abbildung 27 zeigt ein *Kollaborationsdiagramm* zu “Szenario1” des Use Cases “Flurlichtregelung_durch_Anwesenheit”.

Die *Kollaborationsdiagramme* beschreiben die gleiche Informationen wie die Sequenzdiagramme. Welche Darstellungsweise zum Verständnis des Systems herangezogen wird, bleibt der benutzenden Rolle überlassen. Beispielsweise kann es für den Benutzer einfacher sein, das in den Szenarien beschriebene Verhalten in den Sequenzdiagrammen wiederzufinden. Für den Verifizierer mag die horizontale Konsistenz (z. B. zwischen Klassen- und *Kollaborations*-/Sequenzdiagrammen) leichter unter Verwendung der *Kollaborationsdiagramme* überprüfbar sein.

Beispiel *A-Kollaborationsdiagramm*

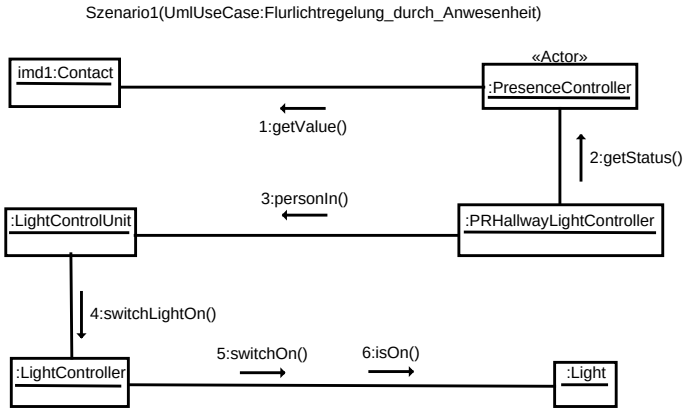


ABBILDUNG 27. Kollaborationsdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit

A-Zustandsdiagramme

Die Komplexität eingebetteter Systeme liegt in ihrem dynamischen Verhalten. Aus diesem Grund ist die Modellierung des Verhaltens sehr wichtig. Die Sequenz- und Kollaborationsdiagramme beschreiben die Interaktion der verschiedenen Objekte. *Zustandsdiagramme* beschreiben das dynamische Verhalten einzelner Klassen.

Ein *Zustandsdiagramm* beschreibt verschiedene Zustände und Transitionen zwischen den Zuständen für die Objekte einer Klasse. Nicht berücksichtigt werden im *Zustandsdiagramm* die Aufrufe von Methoden dieser Klasse, die lediglich zum Setzen oder Abfragen von Werten dienen und keine Zustandsänderung auslösen.

Innerhalb eines Zustands sind verschiedene Aktionen erlaubt: 1. Entry-Aktionen, die beim Eintritt in den Zustand ausgeführt werden, 2. Do-Aktivitäten, die ausgeführt werden, solange ein Zustand aktiv ist, 3. Exit-Aktionen, die beim Verlassen eines Zustands ausgeführt werden und 4. Event-Aktionen, die beim Auftreten eines Events ausgeführt werden. Die Do-Aktivitäten beschreiben teilweise komplexe Methoden, für die eine weitere Modellierung notwendig sein kann. Die Modellierung kann mit Hilfe von Aktivitätsdiagrammen, Pseudo-Code oder informellen Beschreibungen geschehen.

In eingebetteten Systemen müssen Sensorwerte regelmäßig aktualisiert werden. Dies geschieht durch eine laufende Abfrage der Sensorklassen durch die kontrollierenden Klassen. Diese laufenden Abfragen können beispielsweise mit Hilfe von Do-Aktivitäten (Do: each 5 sec /getValue()) oder Event-Aktionen (5secMessage/ getValue()) modelliert werden.

Jedes Zustandsdiagramm wird mit einer informellen Beschreibung erläutert.

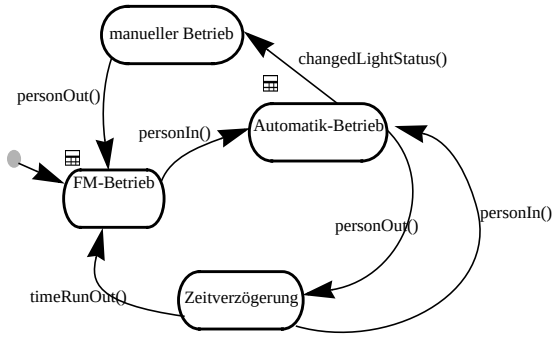
Abbildung 28 zeigt ein *Zustandsdiagramm* für die Klasse “LightControlUnit”.

Beispiel *A-Zustandsdiagramm*

Da das verwendete CASE-Werkzeug StP/UML Entry-, Do- und Exit-Aktionen in separaten State-Tabellen speichert, werden die Aktionen im Anschluss an das *Zustandsdiagramm* aufgeführt. Eine Tabellensymbol an der linken oberen Ecke eines Zustands weist darauf hin, dass es Entry-, Do- und/oder Exit-Aktionen zu diesem Zustand gibt.

Im *Zustandsdiagramm* werden drei Zustände unterschieden: “keine Anwesenheit”, “Anwesenheit” und “Zeitverzögerung”. Die Zustände “keine Anwesenheit” und “Anwesenheit” werden jeweils in zwei Unterzustände verfeinert.

Eine Abfrage von Sensorwerten ist in dieser Klasse nicht notwendig.



Entry-Aktion für State: Automatik-Betrieb

LightController.SwitchLightOn()

Entry-Aktion für State: FM-Betrieb

LightController.SwitchLightOff()

Internal-Aktion für State: FM-Betrieb

Event: fmSwitchOn() Action: LightController.SwitchLightOn()
Event: fmSwitchOff() Action: LightController.SwitchLightOff()

Zustandsdiagramm der Klasse LightControlUnit:
Die Klasse kennt vier Zustände FM-Betrieb, Automatik-Betrieb, manueller Betrieb und Zeitverzögerung. Während des FM-Betriebs ist es dem Hausmeister erlaubt (niemand ist im Flur anwesend) das Flurlicht über das ControlPanelFM einzuschalten. Wenn ein Benutzer im Flurabschnitt anwesend ist, wird unterschieden, ob er den Flurlichtschalter betätigt hat (manueller Betrieb) oder nicht (Automatik-Betrieb). Der manuelle Betrieb wird wieder aufgehoben, wenn alle Benutzer den Flurabschnitt verlassen haben.

ABBILDUNG 28. Zustandsdiagramm der Klasse LightControlUnit mit einem Ausschnitt aus den vorhandenen Entry-, Do- und Exit-Aktionen

A-Aktivitätsdiagramme

Mit Hilfe der *Aktivitätsdiagramme* lassen sich Methoden detaillierter beschreiben, die komplexe Operationen ausführen. Die Funktionalität der Methoden wird mittels der *Aktivitätsdiagramme* in Teilfunktionen zerlegt und beschrieben. Aktivitätsdiagramme sollten nicht verwendet werden, um die Systemfunktionalität zu beschreiben. Dies führt leicht zu einer funktionalen Sichtweise, die sich nur sehr schwer mit einer objektorientierten Strukturierung des Systems verbinden lässt.

Auf Ebene der E-Anforderungen werden die identifizierten komplexen Operationen abstrakt beschrieben. Eine informelle Beschreibung reicht meist aus. Die Aktivitätsdiagramme können auch vollständig durch Pseudo-Code oder informelle Beschreibungen ersetzt werden.

Abbildung 29 zeigt einen Ausschnitt aus einem *A-Aktivitätsdiagramm* für das Teilsystem “Systemmodusermittlung” des Gebäudeautomationssystems.

Beispiel *A-Aktivitätsdiagramm*

UserData::changeValue()

Bei einem Aufruf von changeValue() wird, je nachdem, von welcher Klasse die Instanz gebildet wurde, der Wert auf verschiedene Eigenschaften geprüft und anschließend ein updateValues an die DefinitionOfTempOpModus geschickt.

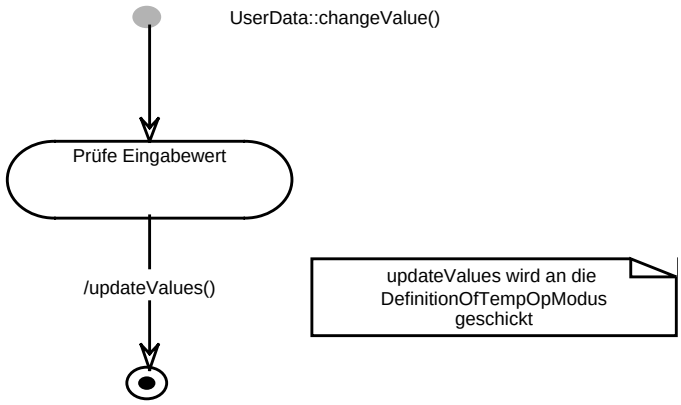


ABBILDUNG 29. Ausschnitt aus einem Aktivitätsdiagramm

Verfolgbarkeitsmatrix-Anforderungen

Die *Verfolgbarkeitsmatrix-Anforderungen* zeigt Beziehungen zwischen den B-Anforderungen und den E-Anforderungen tabellarisch auf. Gegenübergestellt werden die informellen (tabellarischen) B-Anforderungen und die Klassen aus den E-Anforderungen. Ein Kreuz in der Tabelle zeigt, dass eine Klasse an der Realisierung einer informellen B-Anforderung beteiligt ist. Die *Verfolgbarkeitsmatrix* unterstützt eine Überprüfung der Konsistenz zwischen B- und E-Anforderungen und die Durchführung später evtl. auftretender Änderungen. Abbildung 30 zeigt einen Ausschnitt aus einer *Verfolgbarkeitsmatrix-Anforderungen* für die informellen B-Anforderungen des Teilsystems “LichtregelungFlur” des Gebäudeautomationssystems.

Mit Hilfe eines CASE-Werkzeugs, das Verfolgbarkeitsbeziehungen zwischen B- und E-Anforderungen verwaltet, lässt sich die *Verfolgbarkeitsmatrix-Anforderungen* automatisch generieren. Eine Schnittstelle zu einem Requirements-Management Werkzeug wie DOORS ist in diesem Kontext ebenfalls nützlich.

Beispiel *Verfolgbarkeitsmatrix-Anforderungen*

Die *Verfolgbarkeitsmatrix* zeigt beispielsweise, dass an der Realisierung der informellen Anforderung “LichtF-BA-F1” fünf Klassen beteiligt sind: “Light”, “LightControlUnit”, “LightController”, “PRHallwayLightController” und “PresenceController”.

	Light	**LightControlUnit**	**LightController**	**OutdoorTemperatureSensor**	**PRBlindController**	**PRHallwayLightController**	**PRLightController**	**PRTempController**	**PresenceController**
LichtF-BA-F1	X	X	X			X			X
LichtF-BA-F2	X	X	X			X			X

ABBILDUNG 30. Ausschnitt aus der Verfolgbarkeitsmatrix-Anforderungen

A-Verifikation

Das Dokument *A-Verifikation* enthält Hinweise zu und Ergebnisse von durchgeführten Verifikationsaktivitäten auf Ebene der System-Anforderungen. Die folgende Beschreibung beschränkt sich auf die Anwendung einer Checklisten-basierten Inspektionstechnik als Verifikationsaktivität. Das Dokument *A-Verifikation* umfasst dann: 1. eine *A-Checkliste*, 2. eine *Fehlerklassifikation*, 3. eine Liste der gefundenen Fehler (*A-Fehlertabellen*) und 4. eine Beschreibung der Fehlerbehebungsstrategie (*A-Fehlerbehebungsstrategien*).

A-Checkliste

Die *A-Checkliste* unterstützt das Lesen der zu verifizierenden B- und E-Anforderungen und erleichtert das Finden von Problemen in den Dokumenten.

Pro Dokument müssen drei Aspekte überprüft werden: 1. Sind die Dokumentationsrichtlinien erfüllt (d. h. ist das Dokument syntaktisch korrekt), 2. Ist die Qualität des Dokuments ausreichend (d. h. ist das Dokument semantisch korrekt) und 3. Ist das Dokument mit dem Vorgängerdokument konsistent. In der *Checkliste* werden zu jedem Aspekt Fragen gestellt. Werden Fragen in der *Checkliste* mit Nein beantwortet, dann könnte es sich um einen Fehler handeln.

Abbildung 31 zeigt einen Ausschnitt aus einer *A-Checkliste* zur Überprüfung der Use Cases im Use Case-Diagramm.

Beispiel *A-Checkliste*

In diesem Abschnitt sind die Aspekte aufgelistet, die bei der Verifikation der B- und E-Anforderungen zu beachten sind.

Überprüfe jeden Use Case im Use Case Diagramm auf:

Einhaltung der Dokumentationsrichtlinien:

1. Wurde die UML-Syntax eingehalten?
2. Existiert ein Akteur für den Use Case?
3. Existiert zu dem Use Case eine informelle Beschreibung?

Interne Konsistenz der Modelle:

1. *Wenn ein Use Case zu einem anderen in Beziehung steht:* Findet sich die Beziehung auch in den Szenarien, die zu dem Use Case gehören?
2. Sind die Szenarien, die zum Use Case gehören, in der informellen Beschreibung dokumentiert?

Konsistenz mit den informellen B-Anforderungen/Referenzdokumenten

1. *Wenn im Use Case ein Aktuator benutzt wird:* Ist der Aktuator in der Gebäudearchitektur beschrieben?
2. Ist der Akteur des Use Cases in der Gebäudearchitektur / den informellen B-Anforderungen beschrieben?
3. Gibt der Use Case das in den B-Anforderungen beschriebene Verhalten wider?

ABBILDUNG 31. Ausschnitt aus der *A-Checkliste* (B-Anforderungen/E-Anforderungen)

Fehlerklassifikation

Die aufgedeckten Probleme/Fehler werden in Fehlertabellen aufgelistet. Dabei wird jedes Problem/Fehler einer *Fehlerklasse* zugeordnet. Die *Fehlerklassifikation* umfasst spezifische *Fehlerklassen* (z. B. Unvollständigkeit oder falsche Informa-

tion) und unspezifische *Fehlerklassen* (z. B. Inkonsistenz oder Sonstige). Nur wenn ein Fehler keiner spezifischen Fehlerklasse zugeordnet werden kann, wird er einer unspezifischen zugeordnet. Die Erläuterung der Fehlerklassen mit Beispielen hilft, einen existierenden Fehler in eine der vorgegebenen *Klassen* einzuordnen. Abbildung 32 zeigt zwei mögliche *Fehlerklassen* aus einer *Fehlerklassifikation*.

Beispiel *Fehlerklassifikation*

Die während der Verifikation aufgedeckten Probleme/Fehler werden den folgenden Fehlerklassen zugeordnet. Um die Fehler zu klassifizieren, bitte die Fehlerklassen in der angegebenen Reihenfolge durchgehen:

Unvollständigkeit (U):

Es fehlen Informationen zu einem Objekt. Beispielsweise ist eine Klasse nicht im Data-Dictionary beschrieben, obwohl sie im Klassendiagramm vorhanden ist.

Falsche Information (F):

Eine gegebene Information zu einem Objekt ist offensichtlich falsch. Beispielsweise kann ein Zustand in einem Zustandsdiagramm der Klasse Fenster nie erreicht werden.

ABBILDUNG 32. Ausschnitt aus einer Fehlerklassifikation - Spezifische Fehlerklassen

A-Fehlertabellen

In den A-*Fehlertabellen* werden alle *Fehler,* die im Produkt *Problemlösungsdokumentation* gefunden und noch nicht auf nachfolgende Produkte übertragen wurden, aufgelistet. Das bedeutet, dass die *A-Fehlertabelle* Fehler umfasst, die ausschliesslich in der *Problemlösungsdokumentation* enthalten sind. Beispielsweise beinhaltet die *Fehlertabelle* einen *Fehler* im A-Klassendiagramm, der während der Inspektion der System-Anforderungen gefunden wurde. Ein Anforderungsfehler, der die Ursache für einen Entwurfsfehler ist und nur durch den Entwurfsfehler aufgedeckt wurde, wird nicht in der *A-Fehlertabelle* beschrieben.

Ein in der *A-Fehlertabelle* dokumentierter Fehler kann während der Inspektion der System-Anforderungen aufgedeckt worden sein. Er kann aber auch in nachfolgenden Prozessen gefunden worden sein, z. B. beim Erstellen des Entwurfs. Der Prozess, während dessen ein *Fehler* gefunden wurde, ist in der *Tabelle* vermerkt (z. B. “sysanfrevvorb” als Abkürzung für “System-Anforderungen Review vorbereiten”). Die *Fehler* werden mit Hilfe des Produkts und des Teilsystems, in dem sie aufge-

treten sind, gekennzeichnet (*Produkt-Teilsystem*-FNr-*Nummer*). Ein *Fehler* wird durch das betroffene Diagramm und das betroffene Element sowie durch eine kurze Fehlerbeschreibung charakterisiert.

Abbildung 33 zeigt einen Ausschnitt aus einer *A-Fehlertabelle* für das Teilsystem “LichtregelungFlur”.

Beispiel *A-Fehlertabellen*

Die *A-Fehlertabelle* dokumentiert zwei Fehler im Teilsystem “LichtregelungFlur” in den System-Anforderungen. Beide Fehler befinden sich in der Beschreibung der statischen Struktur des Systems (Packagediagramm und Klassendiagramm). Der eine Fehler wurde beim Lesen der Anforderungen gefunden. Der andere Fehler wurde erst bei der Erstellung des UML-Entwurfs gefunden.

FNr	betroffenes Diagramm	betroffenes Element	allgemeine Fehlerbeschreibung	Fehlerklasse	Prozess, währenddessen der Fehler gefunden wurde (Abkürzung)
A-LF-FNr1	Packagestruktur	Package LichtregelungFlur	Package + Beschreibung fehlen	U	sysanfrevvorb
A-LF-FNr2	Klassendiag.	Klasse Haus	falsch geschrieben	F	umlentwbearb

ABBILDUNG 33. Ausschnitt aus der Fehlertabelle Lichtregelung Flur

A-Fehlerbehebungsstrategien

In den *Tabellen zur Dokumentation der Fehlerbehebungsstrategie* wird verdeutlicht, welche Fehler aus den *A-Fehlertabellen* in den System-Anforderungen behoben wurden und wie dies geschehen ist. Außerdem wird in den *Tabellen* die Behebung von Anforderungsfehlern dokumentiert, die die Ursache für Fehler in folgenden Entwicklungsprodukten waren und erst dort entdeckt wurden. Diese Fehler sind in den E- und K-Fehlertabellen dokumentiert. Die *A-Fehlerbehebungsstrategien* geben Aufschluss über den Stand der Rework-Aktivitäten. Anhand der Tabellen kann außerdem entschieden werden, ob eine erneute Verifikation der geänderten B- und E-Anforderungen notwendig ist.

Abbildung 34 zeigt einen Ausschnitt aus einer solchen *Tabelle* für das Teilsystem “LichtregelungFlur”.

Beispiel *A-Fehlerbehebungsstrategien*

In der *Tabelle* werden alle Fehler aus den A-Fehlertabellen der A-Verifikation aufgeführt. Außerdem dokumentiert die *Tabelle* die Behebung eines Anforderungsfehlers, der erst im Kode entdeckt wurde (K-LF-FNr3, K für Kode). Die *Beschreibung der Strategie zur Behebung der Fehler* erlaubt das Nachvollziehen, was sich in den System-Anforderungen geändert hat.

Referenz auf FNr	Strategie zur Behebung des Fehlers	Behebung des Fehlers durchgeführt? (J/N)
A-LF-FNr1	Package “LichtregelungFlur” wurde mit passendem Kommentar eingefügt.	J
A-LF-FNr2	Die Klasse “Haus” wurde in “House” umbenannt.	J
K-LF-FNr3	Aufruf beim Zustandsübergang wurde von SwitchOff() in SwitchOn() geändert.	J

ABBILDUNG 34. Ausschnitt aus der Tabelle zur Dokumentation der Fehlerbehebungsstrategie - Teilsystem Lichtregelung Flur

A-Validation

Das Produkt *A-Validation* umfasst alle Dokumente, die für die *Validation* von zusammengesetzten ausführbaren Produkten (d. h. von Teilsystemen und von dem System) gegen die System-Anforderungen wichtig sind. Dies sind: 1. eine Beschreibung der durchgeführten *Testverfahren* für Teilsystem-, System- und Akzeptanztest, 2. die aufgestellten *Testpläne*, 3. die *Ergebnisse der durchgeführten Tests* sowie 4. eine *Bewertung der Testergebnisse* und die Dokumentation von Fehlverhalten.

Es ist wichtig, Testpläne aufzustellen und durchgeführte Tests und Testergebnisse zu dokumentieren, um gegebenenfalls Tests wiederholen zu können (Regressionstest).

Für das weitere Verständnis werden zunächst die Begriffe Integrationstest, Systemtest und Akzeptanztest definiert. Beim Integrationstest (auch Teilsystemtest genannt) werden ausführbare Teilsysteme unter Zuhilfenahme eines Testrahmens gegen die B-Anforderungen getestet. Ein Test von Teilsystemen gegen die B-Anforderungen ist möglich, wenn die gebildeten Teilsysteme unterschiedliche Verwendungsweisen (z. B. Lichtregelung und Temperaturregelung) des Systems realisieren. Dann entsprechen die B-Anforderungen an das System den Anforderungen an die Teilsysteme. Beim Systemtest wird das ausführbare Gesamtsystem in der Entwicklungsumgebung gegen die B-Anforderungen getestet. Dabei wird sowohl die Übereinstimmung des Systems mit den funktionalen Anforderungen (Funktionstest) als auch die Übereinstimmung mit den nicht-funktionalen Anforderungen überprüft. Beim Akzeptanztest wird das installierte System in der tatsächlichen Systemumgebung durch den Kunden gegen die B-Anforderungen getestet.

Dokumentation der Testverfahren (A-Testverfahren)

Dieses Dokument beschreibt die *Testverfahren*, mit deren Hilfe Testpläne für den Integrations-, den System- und den Akzeptanztest erstellt werden. Es gibt verschiedene *Testverfahren*, die angewendet werden können, z. B. die *Black-box-Testverfahren* Äquivalenzklassen- oder Grenzfallbasiertes Testen. Die Wahl eines Black-box-Testverfahrens erlaubt das Ableiten von Testplänen aus den System-Anforderungen. Das ausführbare Produkt (Teilsystem oder System) wird erst zur Durchführung der Tests benötigt. Ein bestimmtes *Testverfahren* sollte entsprechend des Projekt- und Systemkontexts ausgewählt werden. Abbildung 35 zeigt einen Ausschnitt aus einer *Dokumentation eines Testverfahrens*, das in dem Gebäudeautomationsprojekt verwendet wurde.

Beispiel *Dokumentation eines Testverfahrens*

Beim Integrations- und Systemtest wurde im Gebäudeautomationsprojekt das gleiche Testverfahren verwendet. Es wurde ein Äquivalenzklassentest durchgeführt, der zur Überdeckung der Szenarien führte. Das Testverfahren ließ sich auch für den Integrationstest anwenden, da Teilsysteme nach den Verwendungsweisen des Systems gebildet wurden.

Für jede Ein- und Ausgabevariable wurden gültige und (wenn möglich) ungültige Äquivalenzklassen gebildet. Basierend auf den Szenarien wurden Kombinationen der Äquivalenzklassen der Eingabevariablen gebildet, die zu einem identischen funktionalen Verhalten führen. Ein identisches funktionales Verhalten lässt sich dabei als Äquivalenzklasse einer Ausgabevariable ausdrücken. Die aufgestellten Testpläne führten zur Überdeckung der Szenarien aus den B-Anforderungen und damit zu einer Überdeckung der informellen funktionalen und inversen Anforderungen. Zur Überprüfung der nicht-funktionalen Anforderungen wurden keine Testpläne entwickelt. Abbildung 35 zeigt einen Ausschnitt aus der Dokumentation des Testverfahrens.

Ziel des Systemtests ist es, das System in der Entwicklungsumgebung gegenüber den B-Anforderungen zu validieren. Der Systemtest soll auf den Szenarien aus Kapitel 1.3.1.5.2 basieren, da diese die B-Anforderungen überdecken. Vor Erstellung der Testpläne werden zuerst Eingangs- und Ausgangsvariablen festgelegt, die wiederum in Äquivalenzklassen eingeteilt werden. Die in den Szenarien angegebenen Werte werden anschließend den Äquivalenzklassen zugeordnet.

Im folgenden werden zuerst die Äquivalenzklassen der einzelnen Variablen aufgeführt. Im Anschluss stehen die Pläne der Systemtestfälle, die sich aus der Angabe der Werte für die Eingangs- und Ausgangsvariablen zusammensetzen.

Eingangs- bzw. Ausgangsvariable	gültige Äquivalenzklasse
AnwesenheitskontaktImd1Door1	offen
	geschlossen
LichtstatusFlur	on
	off

ABBILDUNG 35. Ausschnitt aus der Dokumentation eines Testverfahrens

Dokumentation der Testpläne (A-Testpläne)

In der Beschreibung eines Testplans werden die Werte der Eingangsvariablen beschrieben. Außerdem wird notiert, welche Werte die Ausgangsvariablen einnehmen müssen, d. h. wie sich das System verhalten muss.

Abbildung 36 zeigt einen *Testplan* zur Überdeckung des “Szenario1” des Use Cases “Lichtregelung_durch_Anwesenheit”.

Beispiel *Dokumentation der Testpläne*

Die Anzahl der *Testpläne* entspricht der Anzahl der Szenarien. Da die Testpläne die Szenarien überdecken, wurde bei der Beschreibung ein Verweis auf den Szenario-Schritt integriert. Die Werte der Ausgabevariablen zeigen das Verhalten des Systems. Im angegebenen Testplan soll das System beispielsweise das Licht im Flur anschalten (Äquivalenzklasse: LichtStatusFlur: on). Die Äquivalenzklassen für die Eingangs- und Ausgangsvariablen entsprechen den konkreten Werten für den Test, weil der Wertebereich jeder Variable nur aus zwei Elementen besteht.

Test Nummer	Szenario Schritt	Eingangsvariable	Ausgangsvariable	Äquivalenz-klasse	konkreter Wert
SysTest1	1	AnwesenheitskontaktImd1 Door1		offen	offen
		AnwesenheitskontaktImd1 Door2		offen	offen
		AnwesenheitskontaktImd2 Door1		offen	offen
			LichtstatusFlur	off	off
	2	AnwesenheitskontaktImd1 Door1		geschlossen	geschlossen
	3		LichtstatusFlur	on	on

ABBILDUNG 36. Ausschnitt aus der Beschreibung eines Testplans zu Szenario1 - Flurlichtregelung durch Anwesenheit

Dokumentation der Testergebnisse (A-Testergebnisse)

Das Dokument A-Validation erlaubt die *Dokumentation der Testergebnisse* an der gleichen Stelle, an der auch die Testpläne dokumentiert sind. Auf diese Weise können die Testergebnisse mit dem erwartetes Verhalten des Teilsystems oder Systems (abgeleitet aus den B-Anforderungen) leicht miteinander verglichen werden.

Die Testergebnisse beschreiben das Verhalten des (Teil-)Systems beim Testdurchlauf.

Abbildung 37 zeigt eine mögliche *Dokumentation der Ergebnisse eines durchgeführten Tests* anhand des oben aufgeführten Testplans.

Beispiel *Dokumentation der Testergebnisse*

Testergebnis SystemTest1:

AnwesenheitskontaktImd1 Door 1 = offen

AnwesenheitskontaktImd2 Door 2 = offen

Lichtstatus Flur = off

AnwesenheitsheitskontaktImd1 Door 1 = geschlossen

Lichtstatus Flur = off

ABBILDUNG 37. Dokumentation zum SystemTest1 des Teilsystems Lichtregelung - Flur

A-Ergebnisbewertung

Nachdem die Tests anhand der Testpläne durchgeführt und die *Testergebnisse* dokumentiert sind, kann die Auswertung der Tests erfolgen. Dafür werden die *Testergebnisse* mit dem im Testplan dokumentierten Ausgabeverhalten des (Teil)-Systems verglichen. Die Ergebnisse der Auswertung, d. h. entdeckte Fehlverhalten, werden mit Hilfe einer Tabelle dokumentiert.

Abbildung 38 zeigt einen Ausschnitt aus einer *Tabelle zur Dokumentation von gefundenen Fehlverhalten*.

Beispiel *A-Ergebnisbewertung*

Die Fehlverhalten werden zur leichteren Referenzierung durchnummeriert, wobei die Nummer den Validationsschritt (IT-Integrationstest, ST-Systemtest, AT-Akzeptanztest und TB-Test im Betrieb) und das Teilsystem (LF-LichtregelungFlur) enthält. Die Angabe des Schritts im Testplan erlaubt das leichtere Nachvollziehen der aufgedeckten Fehlverhalten.

FV-Nr	Verweis auf Test	Schritt imTest	Beschreibung des Fehlverhaltens	Verweis auf Fehler-Nr.
ST-LF-FVNr1	SystemTest1	Schritt 3	Das Licht wird trotz Personenanwesenheit nicht angeschaltet.	K-LF-FNr 45

ABBILDUNG 38. Ausschnitt aus der Tabelle zur Bewertung der Testergebnisse

Die beim Test entdeckten Fehlverhalten lassen sich auf Fehler im Kode zurückführen und werden in der K-Fehlertabelle im Dokument K-Verifikation dokumentiert. In die Tabelle zur Bewertung der Testergebnisse wird ein Verweis auf die Fehlernummern ergänzt.

Namenskonventionen

Die Einführung von *Namenskonventionen* bereits auf Ebene der System-Anforderungen gewährleistet die Namensgleichheit zu späteren Entwicklungsprodukten und unterstützt damit die Verständlichkeit und Änderbarkeit der System-Dokumentation. Außerdem erleichtert sie eine Abstimmung im Team. Abbildung 39 zeigt mögliche *Namenskonventionen* für System-Anforderungen.

Beispiel *Namenskonventionen*

- Bezeichner tragen natürlichsprachliche oder problemnahe Namen oder sind verständliche Abkürzungen solcher Namen.
- Kein Name beginnt mit einem Underscore (_).
- Zwei Bezeichner dürfen sich in einem Sichtbarkeitsbereich nicht ausschließlich durch Unterschiede in der Groß-/Kleinschreibung unterscheiden.
- Alle Bezeichner werden der englischen Sprache entnommen.
- Bilden mehrere Teilworte einen Namen, so sollten zur optischen Trennung der Teilworte einzelne Großbuchstaben verwendet werden. Underscores werden nicht eingesetzt.
- Klassennamen beginnen mit einem Großbuchstaben.
- Alle anderen Namen beginnen mit einem Kleinbuchstaben.
- Außer zur optischen Trennung enthalten Bezeichner keine weiteren Großbuchstaben.
- Methoden, die zum Lesen bzw. Setzen von Attributen benutzt werden, werden als solche gekennzeichnet. Sie erhalten einen entsprechenden Präfix (set, get).
- Methoden, die einen boolschen Wert zurückliefern, erhalten is oder has als Präfix.

ABBILDUNG 39. Namenskonventionen auf Ebene der System-Anforderungen

3.3 Rollen

Im Prozess *System-Anforderungen bearbeiten* sind die Rollen *Anforderungsingenieur*, *Kunde*, *Domänenexperte*, *Verifizierer*, *Validierer* und *Qualitätsmanager* involviert. Ihre Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 40. Folgende wesentliche Tätigkeiten sind im Prozess *System-Anforderungen erstellen* mit den Rollen verbunden:

- ***Anforderungsingenieur.*** Die Aufgaben des Anforderungsingenieurs sind, die B- und E-Anforderungen sowie die Verfolgbarkeitsmatrix zu erstellen. Dies sollte in enger Abstimmung mit dem Kunden/Benutzer erfolgen. Nach einer Verifikation bzw. Validation ist der Anforderungsingenieur für erforderliche Korrekturen, Änderungen und Erweiterungen entsprechend der Verifikations- bzw. Validationsergebnisse verantwortlich.
- ***Kunde.*** Die Aufgabe des Kunden/Auftraggebers ist, bei der Erstellung der System-Anforderungen mitzuwirken. Aus Gründen der Vereinfachung wird hier der Kunde mit dem späteren Benutzer/Anwender des Systems gleichgesetzt. Dies ist in der Praxis nicht immer gegeben. In einem solchen Fall kann es durchaus sinnvoll sein, potentielle Benutzer/Anwender bei der Erstellung der System-Anforderungen einzubeziehen.

- ***Domänenexperte.*** Die Aufgaben des Domänenexperten sind, das Produkt *Domänenwissen* bereitzustellen und zu warten sowie für Auskünfte über die Domäne bereitzustehen.
- ***Verifizierer.*** Die Aufgaben des Verifizierers sind, die Inspektion der System-Anforderungen zu organisieren, diese in Kooperation mit dem Anforderungsingenieur durchzuführen und die Ergebnisse zu dokumentieren.
- ***Validierer.*** Die Aufgabe des Validierers im Prozess *System-Anforderungen bearbeiten* ist, Testfälle zu entwickeln und zu dokumentieren.
- ***Qualitätsmanager.*** Die Aufgabe des Qualitätsmanagers ist, entsprechend den längerfristigen Bedürfnissen und Erwartungen einer Software-Entwicklungsorganisation Wissen/Erfahrung in geeigneter Form zu verwalten, einzelnen Projekten auf Anfrage zur Wiederverwendung zur Verfügung zu stellen und kontinuierlich aufgrund gemachter Projekterfahrungen zu verbessern.
  Wissen/Erfahrung werden vom Qualitätsmanager im Prozess *System-Anforderungen bearbeiten* in Form von Checklisten, Namenskonventionen, Fehlerklassifikationen etc. zur Verfügung gestellt. Diese sind zum Teil Bestandteil von Produkten, für die andere Rollen verantwortlich sind. Eigenverantwortlich ist der Qualitätsmanager hier für das Produkt *Namenskonventionen*.

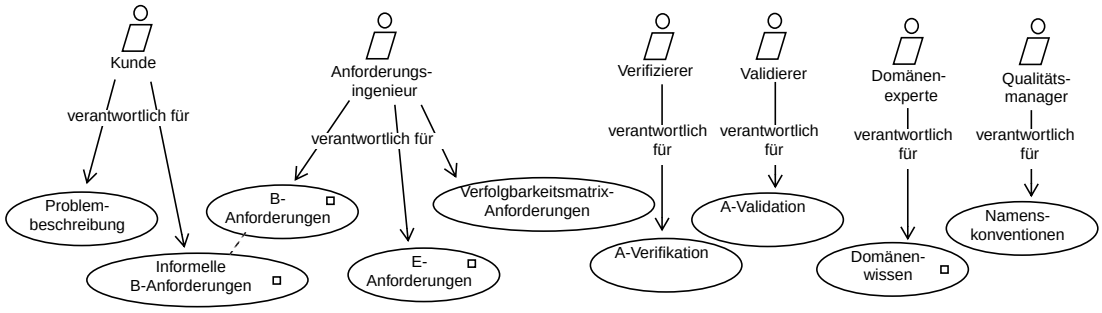


ABBILDUNG 40. Ressourcen und Produkte des Prozesses *System-Anforderungen bearbeiten*

3.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 41 dargestellt.

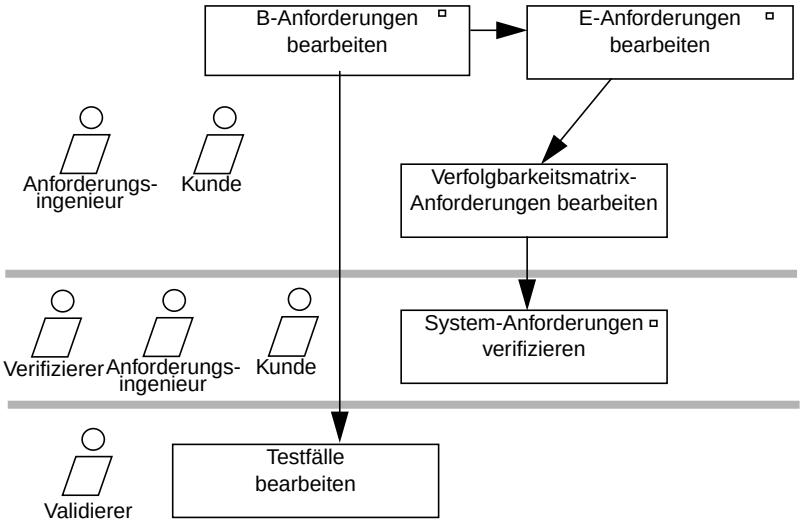


ABBILDUNG 41. Kontrollfluss für den Prozess *System-Anforderungen bearbeiten*

3.5 Prozess "B-Anforderungen bearbeiten"

Im Prozess *B-Anforderungen bearbeiten* werden aus der *Problembeschreibung*, den *informellen (tabellarischen) B-Anforderungen* und dem *Domänenwissen A-Use Cases* und *Szenarien* erarbeitet.

Das System selbst wird auf Ebene der B-Anforderungen als Black-box betrachtet. Das Verhalten des Systems wird mittels Beziehungen zwischen Ein- und Ausgabegrößen beschrieben.

Der Prozess *B-Anforderungen bearbeiten* geschieht in Interaktion mit dem Benutzer/Kunden. Hauptziel ist, ein gemeinsames Verständnis der Aufgabenstellung und des Anwendungsgebiets zwischen dem Anforderungsingenieur und dem Kunden/Benutzer zu schaffen.

Abbildung 42 zeigt den Prozess *B-Anforderungen bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

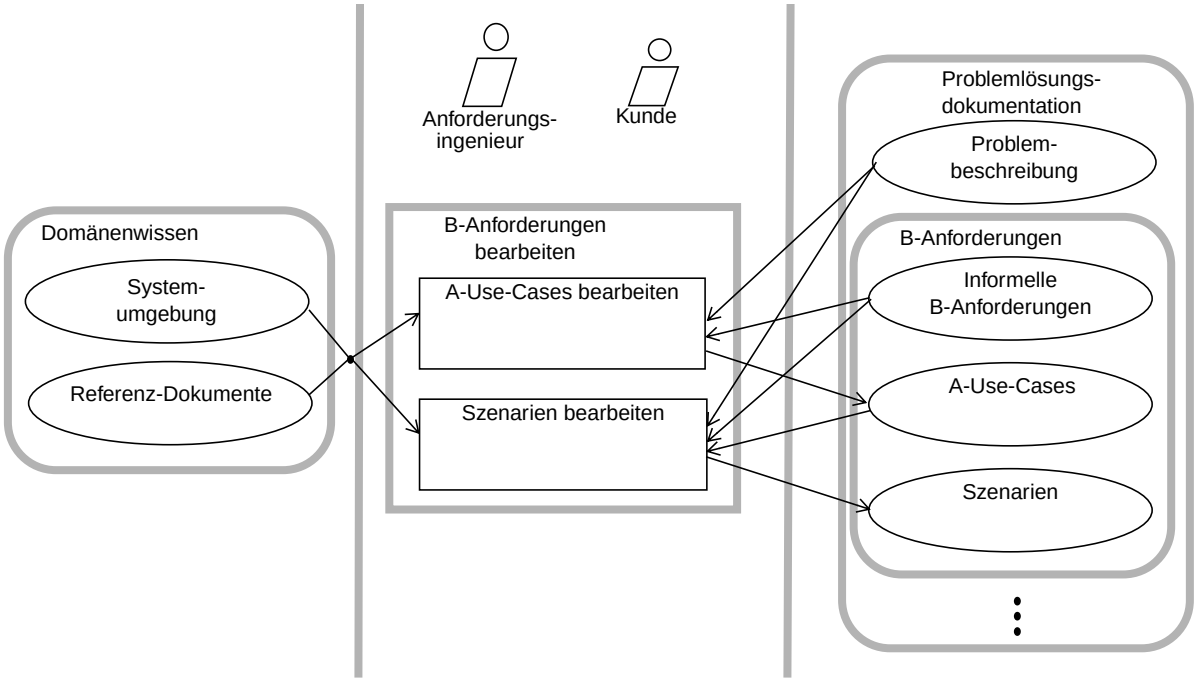


ABBILDUNG 42. Verwendete Produkte des Prozesses *B-Anforderungen bearbeiten*

Folgende Produkte werden konsumiert:

- Systemumgebung. Dieses Dokument beschreibt spezielle Eigenschaften der Domäne.
- Referenz-Dokumente. Diese Dokumente präzisieren einzelne Systemaspekte oder schildern Erfahrungen aus ähnlichen Projekten.
- Problembeschreibung. Dieses Dokument beschreibt das zu lösende Problem und das Anwendungsgebiet informell. Es wird vom Kunden vorgegeben.
- Informelle B-Anforderungen. Dieses Dokument beschreibt die funktionalen, nicht-funktionalen und inversen Anforderungen an das zu entwickelnde System in tabellarischer Form und in natürlicher Sprache. Es verfeinert die Problembeschreibung.

Folgendes Produkt wird modifiziert:

- B-Anforderungen. Neben den informellen (tabellarischen) B-Anforderungen enthält das Dokument A-Use Cases und Szenarien. Die A-Use Cases beschreiben die möglichen Verwendungsweisen des zu entwickelnden Systems. Die Szenarien verfeinern die A-Use Cases. Sie geben Beispiele für konkrete Interaktionssequenzen zwischen den Akteuren und dem System.

Prozess *"A-Use Cases bearbeiten"*

Im Prozess *A-Use Cases bearbeiten* wird ein Use Case-Diagramm erstellt und jeder Use Case informell beschrieben. Für die Erstellung eines Use Case-Diagramms und von einzelnen Use Cases sind u. a. folgende Fragen wichtig:

1. Welche Akteure gibt es im System und wie benutzen sie das System?
   Zur Beantwortung dieser Frage sollte die Beschreibung der Sensoren und Aktuatoren aus der Systemumgebung herangezogen werden. Außerdem werden evtl. in den Beschreibungen des erwarteten Verhaltens des Systems (Problembeschreibung, informelle B-Anforderungen) verschiedene Akteure eingeführt (z. B. ein Hausmeister und ein Benutzer eines Raums). Außerdem wird in den Dokumenten beschrieben, wie die Akteure das System benutzen.
2. Welche Aufgaben soll das System erfüllen?
   Diese Frage kann einerseits unter Verwendung der Problembeschreibung und der informellen B-Anforderungen beantwortet werden. Andererseits sollte festgestellt werden, welche Größen gemessen und vom System kontrolliert werden sollen. Aus diesen Größen lässt sich ableiten, was das System leisten soll. Zur Bestimmung der Größen kann wieder die Beschreibung der Sensoren und Aktuatoren aus der Systemumgebung herangezogen werden.

Jeder Use Case wird kurz mit Hilfe des bereitgestellten Schemas beschrieben.

Use Cases können in Teil-Use Cases verfeinert werden, z. B. wenn mehrere Use Cases in einem Teil vollständig gleich sind, kann der gleiche Teil in einem separaten Use Case beschrieben werden. Die anderen Use Cases benutzen dann den ausgelagerten Use Case. Erfahrungen aus dem Gebäudeautomationsprojekt zeigen jedoch, dass eine Verfeinerung von Use Cases durch die Entwickler als eine funktionale Zerlegung des Systems verstanden werden kann. Mit einer Verfeinerung sollte deshalb vorsichtig umgegangen werden.

Folgende Produkte werden im Prozess *A-Use Cases bearbeiten* konsumiert:

- Systemumgebung.

- Referenz-Dokumente.
- Problembeschreibung.
- Informelle B-Anforderungen.

Folgendes Produkt wird produziert:

- A-Use Cases.

Prozess *“Szenarien bearbeiten”*

Die einzelnen Use Cases werden mittels Szenarien konkretisiert. Die Szenarien sollten dabei die informellen (tabellarischen) B-Anforderungen überdecken. Bei der Beschreibung der Szenarien sollte Folgendes beachtet werden:

1. Zu jedem Szenario sollten die Vorbedingungen beschrieben werden, die erfüllt sein müssen, damit das Szenario eintreten kann.
2. Jedes Szenario beginnt mit einer Aktion des Akteurs.
3. Jedes Szenario sollte genau eine Interaktionssequenz zwischen Akteur und System darstellen, d. h.:

   1. Die Vorbedingungen des Szenarios sind so zu konstruieren, dass nur eine Systemreaktion ausgelöst wird (z. B. nicht Temperaturregelung und Lichtregelung gleichzeitig).

   2. Das Szenario endet, wenn das System eine Reaktion zeigt (z. B. Licht wird angeschaltet).
4. Das Szenario sollte konkrete Werte enthalten.

Folgende Produkte werden in dem Prozess *Szenarien bearbeiten* konsumiert:

- Systemumgebung.
- Referenz-Dokumente.
- Problembeschreibung.
- Informelle B-Anforderungen.
- A-Use Cases.

Folgendes Produkt wird produziert:

- Szenarien.

3.6 Prozess “E-Anforderungen bearbeiten”

Im Prozess *E-Anforderungen bearbeiten* werden die *B-Anforderungen* verfeinert und in eine für die Entwickler vertraute Notation überführt. Die *Namenskonventionen* geben Richtlinien für Bezeichner und gewährleisten damit eine einheitliche Namensgebung in allen weiteren Entwicklungsprodukten. Bei der Verfeinerung der *B-Anforderungen* hilft Wissen über die Domäne. Es wird eine Struktur des Systems definiert (*A-Klassendiagramme bearbeiten*) und beschrieben (*Data Dictionary bearbeiten*) und das Verhalten des Systems auf eine Interaktion zwischen den Komponenten im Systems abgebildet (*A-Sequenzdiagramme/A-Kollaborationsdiagramme bearbeiten* und *A-Zustandsdiagramme bearbeiten*). Des Weiteren werden *Aktivitätsdiagramme* (*A-Aktivitätsdiagramme bearbeiten*) erstellt, mit denen einzelne Operationen verfeinert werden.

Abbildung 43 zeigt den Prozess *E-Anforderungen bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

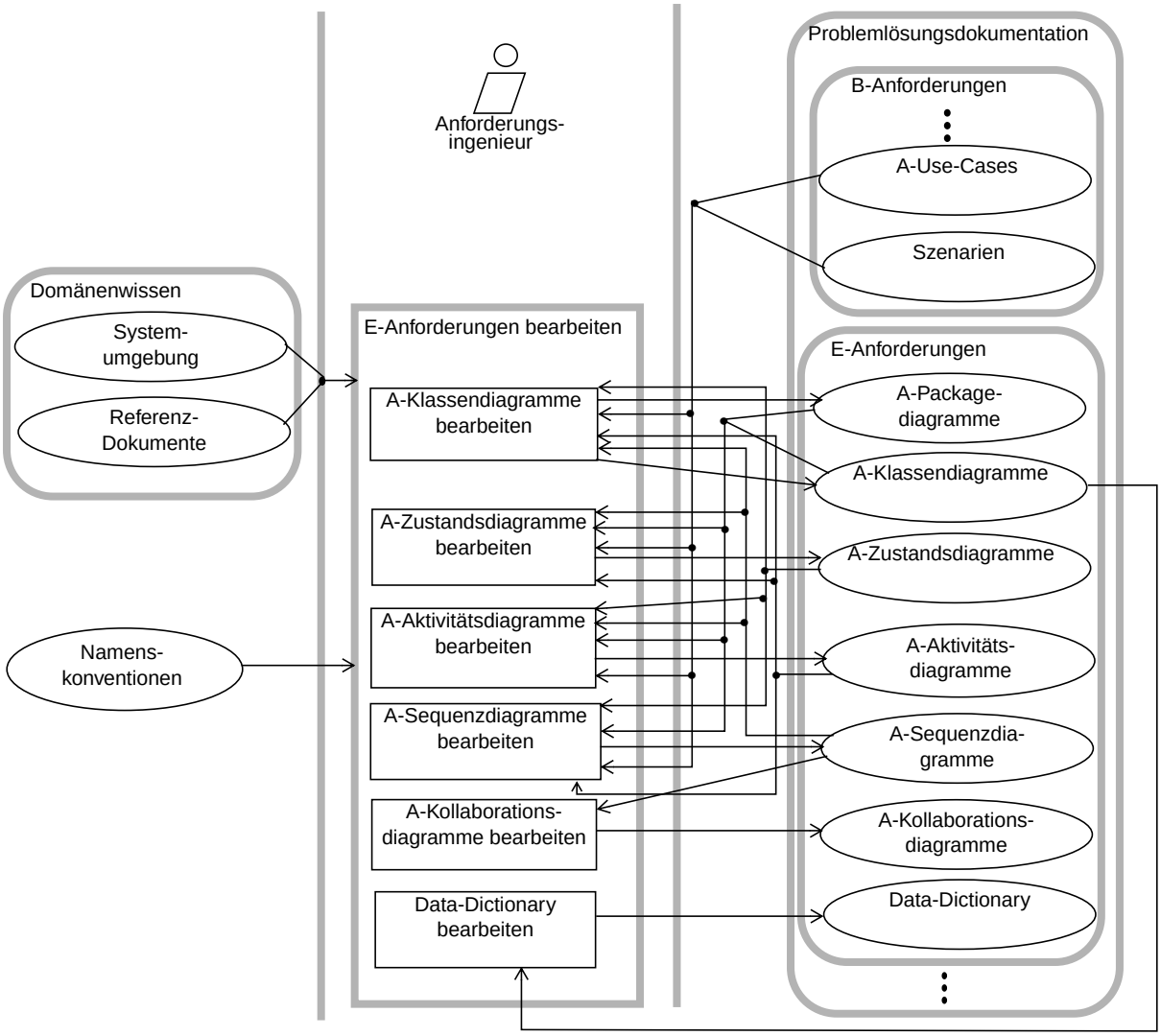


ABBILDUNG 43. Verwendete Produkte des Prozesses *E-Anforderungen bearbeiten*

Folgende Produkte werden im Prozess *E-Anforderungen bearbeiten* konsumiert:

- Systemumgebung.
- Referenz-Dokumente.
- A-Use Cases.
- Szenarien.
- Namenskonventionen.

Folgendes Produkt wird produziert:

- E-Anforderungen. Die E-Anforderungen beschreiben das Verhalten des zu entwickelnden Systems aus Sicht der Entwickler. Es umfasst verschiedene UML-Diagramme. Die A-Klassen- und Packagediagramme beschreiben die Struktur des zu entwickelnden Systems. Das Data-Dictionary beschreibt die Klassen, Attribute, Methoden und Beziehungen aus den Klassendiagrammen. Die A-Sequenz- und A-Kollaborationsdiagramme beschreiben das Verhalten des Systems durch die Interaktion von Objekten. Die A-Zustandsdiagramme beschreiben das Verhalten einzelner Klassen. Die A-Aktivitätsdiagramme verfeinern einzelne Operationen.

Die verschiedenen Teilprozesse des Prozesses *E-Anforderungen bearbeiten* beeinflussen sich gegenseitig und lassen sich nicht unabhängig voneinander durchführen: Wird z. B. eine Nachricht an ein Objekt einer Klasse in einem Sequenzdiagramm geschickt (*A-Sequenzdiagramme bearbeiten*), so muss diese Nachricht auch als Methode dieser Klasse in das Klassendiagramm (*A-Klassendiagramme bearbeiten*) und evtl. in das Zustandsdiagramm (*A-Zustandsdiagramme bearbeiten*) eingefügt werden. Die Konsistenz zwischen den verschiedenen Diagrammen muss gewährleistet werden. Zu jedem Teilprozess werden im Weiteren Beispiele für Konsistenzbedingungen angegeben.

Prozess *"A-Klassendiagramme bearbeiten"*

Während des Prozesses *A-Klassendiagramme bearbeiten* werden Teilsysteme und Klassen innerhalb der Teilsysteme bestimmt und in Packagediagrammen und Klassendiagrammen festgehalten. Außerdem werden Beziehungen zwischen Klassen erarbeitet. Beziehungen zwischen Klassen in verschiedenen Teilsystemen werden durch Beziehungen zwischen Packages verdeutlicht. Für Teilsysteme und Klassen wird eine kurze Erläuterung eingefügt.

Klassen, die in mehreren Teilsystemen verwendet werden, sollten zu einem ausgezeichneten Teilsystem gehören, in dem Methoden und Attribute der Klasse bearbeitet werden. In anderen Teilsystemen dürfen die Klassen verwendet, aber nicht bearbeitet werden. Diese Beschränkung ist sinnvoll, um Inkonsistenzen zwischen Teilsystemen zu vermeiden und eindeutige Schnittstellen zu bilden.

Klassen innerhalb eines Teilsystems werden anhand der Begriffe aus der Anwendung gebildet (Referenzdokument: Systemumgebung). Attribute und Methoden werden anhand des gewünschten Verhaltens bestimmt (Referenzdokument: A-Use Cases und Szenarien). Die Namenskonventionen geben Richtlinien für Benennungen von Bezeichnern (z. B. Bezeichner werden der engl. Sprache entnommen). Die Konsistenz der anderen UML-Diagramme mit den Klassendiagrammen muss sichergestellt sein. Dafür muss u. a. geprüft werden, ob:

- die Attribute und Methoden im Zustandsdiagramm im Klassendiagramm definiert sind.
- Klassen, deren Objekte im Sequenzdiagramm miteinander kommunizieren, eine Beziehung im Klassendiagramm haben.
- jedes Objekt im Sequenzdiagramm einer Klasse im Klassendiagramm zugeordnet werden kann.

Folgende Produkte werden im Prozess *A-Klassendiagramme bearbeiten* konsumiert:

- Systemumgebung.
- Referenz-Dokumente.
- Namenskonventionen.
- A-Use Cases.
- Szenarien.
- A-Sequenzdiagramme.
- A-Zustandsdiagramme.
- A-Aktivitätsdiagramme.

Folgende Produkte werden produziert:

- A-Packagediagramme.
- A-Klassendiagramme.

Prozess *"Data-Dictionary bearbeiten"*

Die Klassen sowie deren Attribute, Methoden und Assoziationen/Aggregationen werden informell beschrieben.

Das CASE-Werkzeug StP/UML erlaubt aus den Klassendiagrammen und aus angelegten Beschreibungen ein *Data-Dictionary* automatisch zu generieren.

Folgendes Produkt wird im Prozess *Data-Dictionary bearbeiten* konsumiert:

- A-Klassendiagramme.

Folgendes Produkt wird produziert:

- Data-Dictionary.

Prozess *"A-Sequenzdiagramme bearbeiten"*

Für jedes Szenario aus den B-Anforderungen wird ein Sequenzdiagramm gebildet. Das Verhalten des Systems wird als Interaktion von Objekten der Klassen aus dem Klassendiagramm beschrieben. Die Namenskonventionen geben Richtlinien für Benennungen von Bezeichnern (z. B. Bezeichner werden der engl. Sprache entnommen). Die Konsistenz der anderen UML-Diagramme mit dem Sequenzdiagramm muss sichergestellt sein. Dafür muss u. a. geprüft werden, ob:

- Methoden einer Klasse im Klassendiagramm im Sequenzdiagramm verwendet werden.
- Events im Zustandsdiagramm als Nachrichten modelliert sind, die ein Objekt einer Klasse geschickt bekommt.
- Aktionen im Zustandsdiagramm als Nachrichten modelliert sind, die ein Objekt einer Klasse verschickt.

Folgende Produkte werden im Prozess *A-Sequenzdiagramme bearbeiten* konsumiert:

- Systemumgebung.
- Referenz-Dokumente.
- Namenskonventionen.
- A-Use Cases.
- Szenarien.
- A-Packagediagramme.

- A-Klassendiagramme.
- A-Zustandsdiagramme.
- A-Aktivitätsdiagramme.

Folgendes Produkt wird produziert:

- A-Sequenzdiagramme.

Prozess *“A-Kollaborationsdiagramme bearbeiten”*

Im Prozess *A-Kollaborationsdiagramme bearbeiten* werden aus den Sequenzdiagrammen Kollaborationsdiagramme abgeleitet.

Mit CASE-Werkzeugen (z. B. StP/UML) können aus den Sequenzdiagrammen Kollaborationsdiagramme automatisch generiert werden. Die Generierung sollte geschehen, wenn sich die Sequenzdiagramme in einem stabilen Zustand befinden.

Folgendes Produkt wird in dem Prozess *A-Kollaborationsdiagramme bearbeiten* konsumiert:

- A-Sequenzdiagramme.

Folgendes Produkt wird produziert:

- A-Kollaborationsdiagramme.

Prozess *“A-Zustandsdiagramme bearbeiten”*

Für jede Klasse mit dynamischen Verhalten wird ein Zustandsdiagramm angelegt. Das Diagramm zeigt die verschiedenen Zustände und Bedingungen für Zustandstransitionen. Die Namenskonventionen geben Richtlinien für Benennungen von Bezeichnern (z. B. Bezeichner werden der engl. Sprache entnommen). Die Konsistenz der anderen UML-Diagramme mit den Zustandsdiagrammen muss sichergestellt sein. Dafür muss u. a. geprüft werden, ob:

- alle Methoden, die im Sequenzdiagramm von einem Objekt einer Klasse verschickt werden, im Zustandsdiagramm der Klasse als Aktionen auftreten.
- Beziehungen zwischen Klassen im Klassendiagramm im Zustandsdiagramm genutzt werden.

Folgende Produkte werden im Prozess *A-Zustandsdiagramme bearbeiten* konsumiert:

- Systemumgebung.
- Referenz-Dokumente.
- Namenskonventionen.
- A-Use Cases.
- Szenarien.
- A-Packagediagramme.
- A-Klassendiagramme.
- A-Sequenzdiagramme.
- A-Aktivitätsdiagramme.

Folgendes Produkt wird produziert:

A-Zustandsdiagramme.

Prozess *"A-Aktivitätsdiagramme bearbeiten"*

Jede komplexe Operation einer Klasse wird mit Hilfe eines Aktivitätsdiagramms verfeinert. Anstelle von Aktivitätsdiagrammen können die Operationen auch mit Hilfe von Pseudo-Code oder informellen Beschreibungen erläutert werden. Das Aktivitätsdiagramm erlaubt die Verfeinerung einer Aktivität in Teilaktivitäten. Die Namenskonventionen geben Richtlinien für Benennungen von Bezeichnern (z. B. Bezeichner werden der engl. Sprache entnommen). Die Konsistenz der anderen UML-Diagramme mit den Aktivitätsdiagrammen muss sichergestellt sein. Dafür muss u. a. geprüft werden, ob:

- für komplexe Operationen im Klassendiagramm ein Aktivitätsdiagramm existiert.
- für komplexe Operationen im Zustandsdiagramm ein Aktivitätsdiagramm existiert.
- Beziehungen zwischen Klassen im Klassendiagramm im Aktivitätsdiagramm genutzt werden.

Folgende Produkte werden im Prozess *A-Aktivitätsdiagramme bearbeiten* konsumiert:

- Systemumgebung.

- Referenz-Dokumente.
- Namenskonventionen.
- A-Use Cases.
- Szenarien.
- A-Packagediagramme.
- A-Klassendiagramme.
- A-Zustandsdiagramme.
- A-Sequenzdiagramme.

Folgendes Produkt wird produziert:

- A-Aktivitätsdiagramme.

3.7 Prozess “Verfolgbarkeitsmatrix-Anforderungen bearbeiten”

Im Prozess *Verfolgbarkeitsmatrix-Anforderungen bearbeiten* wird eine Beziehung zwischen den informellen (tabellarischen) B-Anforderungen und den Klassen aus den E-Anforderungen hergestellt. Für jede Klasse, die an der Realisierung einer informellen B-Anforderung beteiligt ist, wird ein Kreuz in der Matrix eingefügt.

Mit Hilfe von CASE-Werkzeugen (z. B. StP/UML) ist eine Generierung der Verfolgbarkeitsmatrix aus vorhandenen Verfolgbarkeitsbeziehungen möglich. Um Verfolgbarkeitsbeziehungen anzulegen, werden beispielsweise Beziehungen zwischen informellen B-Anforderungen und Klassen per Mausklick gesetzt.

Abbildung 44 zeigt den Prozess *Verfolgbarkeitsmatrix-Anforderungen bearbeiten* mit konsumierenden und produzierenden/modifizierenden Produkten.

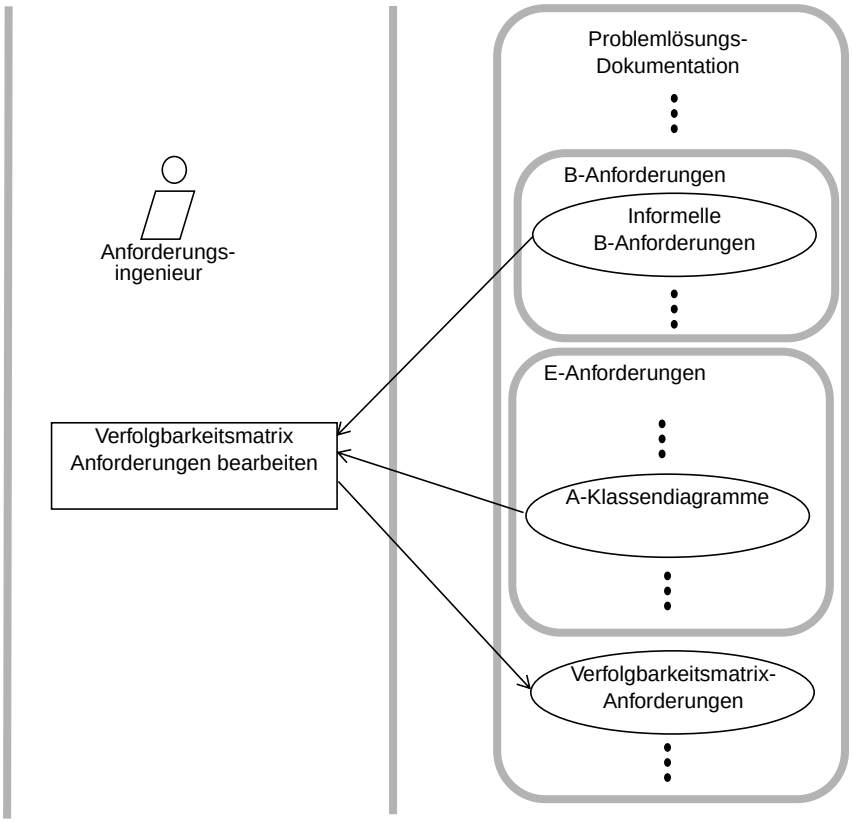


ABBILDUNG 44. Verwendete Produkte des Prozesses *Verfolgbarkeitsmatrix-Anforderungen bearbeiten*

Folgende Produkte werden konsumiert:

- Informelle B-Anforderungen.
- A-Klassendiagramme.

Folgendes Produkt wird produziert:

- Verfolgbarkeitsmatrix-Anforderungen.

3.8 Prozess “System-Anforderungen verifizieren”

Im Prozess *System-Anforderungen verifizieren* werden die A-Use Cases und Szenarien gegen die informellen B-Anforderungen, die Problembeschreibung, die Systemumgebung und die Referenz-Dokumente verifiziert. Die E-Anforderungen werden gegen die B-Anforderungen, die Systemumgebung, die Referenz-Dokumente und die Namenskonventionen verifiziert. Außerdem wird die Verfolgbarkeitsmatrix-Anforderungen überprüft. Anleitungen für und Ergebnisse der Verifikation werden im Dokument A-Verifikation dokumentiert.

Im folgenden wird sich auf die Inspektion als Verifikationsaktivität eingeschränkt.

Ein Inspektionsprozess umfasst drei Teilprozesse: 1. Lesen des zu verifizierenden Dokuments (*System-Anforderungen Review vorbereiten*), 2. Durchführung einer Inspektionssitzung zur Besprechung aufgedeckter Fehler (*Reviewsitzung System-Anforderungen*) und 3. Überarbeiten des verifizierten Dokuments zur Behebung von Fehlern (*Rework System-Anforderungen*).

Abbildung 45 zeigt den Prozess *System-Anforderungen verifizieren* mit konsumierenden, produzierenden und modifizierenden Produkten.

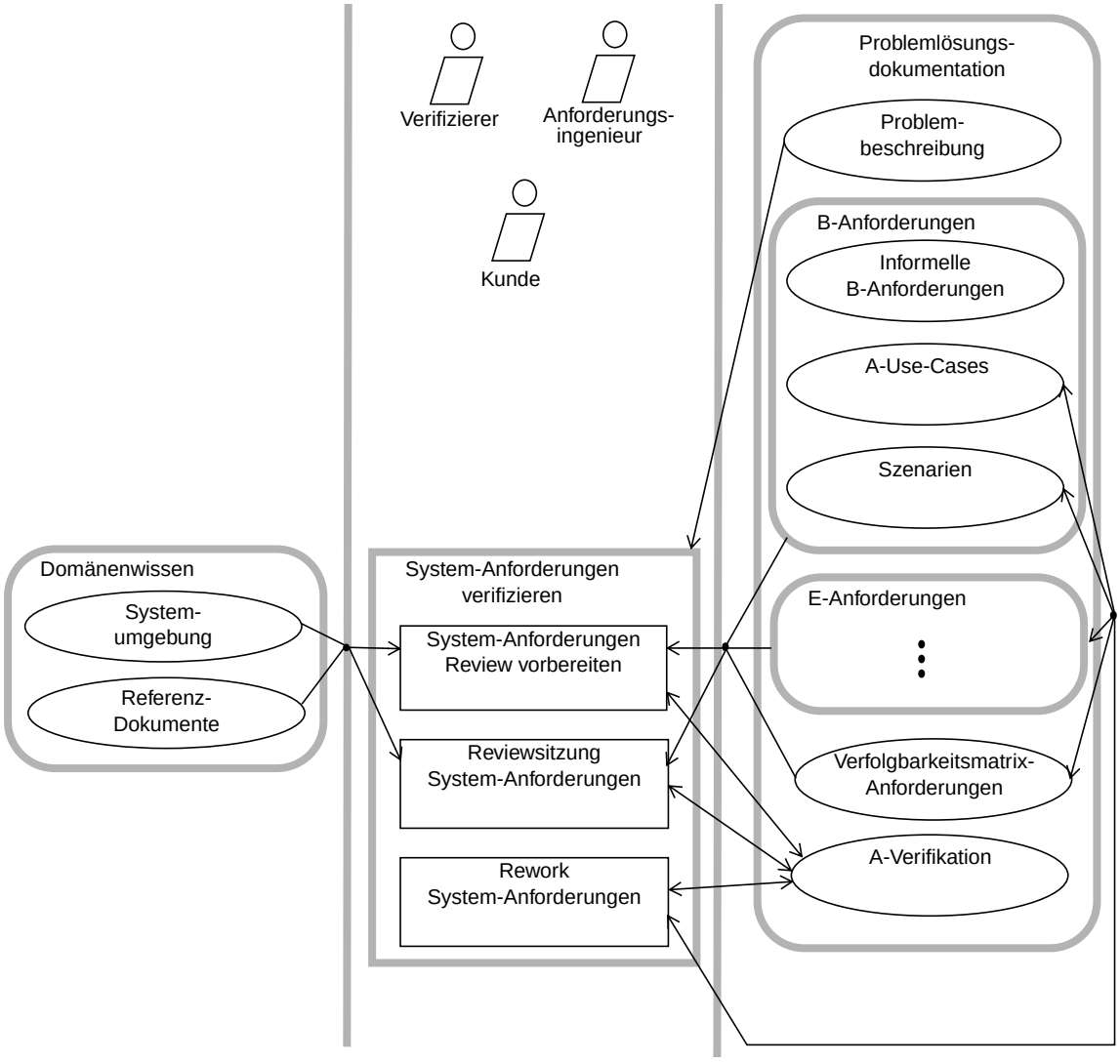


ABBILDUNG 45. Verwendete Produkte des Prozesses *System-Anforderungen verifizieren*

Folgende Produkte werden im Prozess *System-Anforderungen verifizieren* konsumiert:

- Problembeschreibung.
- Systemumgebung.
- Referenz-Dokumente.
- Informelle B-Anforderungen (Teilprodukt der B-Anforderungen).

Folgende Produkte werden modifiziert:

- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.

Prozess *“System-Anforderungen Review vorbereiten”*

Im Prozess *System-Anforderungen Review vorbereiten* sollen Probleme in den A-Use Cases, den Szenarien, den E-Anforderungen und in der Verfolgbarkeitsmatrix-Anforderungen aufgedeckt und in einer Fehlertabelle dokumentiert werden. Hierzu werden die Dokumente mit Hilfe einer Checkliste überprüft, die im Dokument A-Verifikation dokumentiert ist. Werden Fragen in der Checkliste mit “Nein” beantwortet, dann könnte es sich um einen Fehler handeln. Gelesen wird von den Verifizierern.

Ergebnis dieses Prozesses ist eine Liste mit Problemen im Dokument (d. h. möglichen Fehlern). Jeder potentielle Fehler ist dabei einer Fehlerklasse zuzuordnen. Die potentielle Fehlerliste wird im Dokument A-Verifikation dokumentiert.

Im Prozess *System-Anforderungen Review vorbereiten* werden folgende Produkte konsumiert:

- Problembeschreibung.
- Systemumgebung.
- Referenz-Dokumente.
- B-Anforderungen.
- E-Anforderungen.

- Verfolgbarkeitsmatrix-Anforderungen.

Folgendes Produkt wird modifiziert:

- A-Verifikation.

Prozess *"Reviewsitzung System-Anforderungen"*

Im Prozess *Reviewsitzung System-Anforderungen* werden die beim Lesen gefundenen Fehler besprochen. An der Inspektionssitzung nehmen Anforderungsingenieure und Verifizierer teil. Die Aufgabe der Verifizierer ist es nicht, die erkannten Fehler zu lösen, sondern lediglich, diese während der Sitzung aufzuzeigen. Die Verifizierer führen durch die Liste der potentiellen Fehler. Ergebnis der Inspektionssitzung ist eine überarbeitete Fehlerliste, die im Dokument A-Verifikation festgehalten wird.

Im Prozess *System-Anforderungen Review vorbereiten* werden folgende Produkte konsumiert:

- Problembeschreibung.
- Systemumgebung.
- Referenz-Dokumente.
- B-Anforderungen.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.

Folgendes Produkt wird modifiziert:

- A-Verifikation.

Prozess *"Rework System-Anforderungen"*

Im Prozess *Rework System-Anforderungen* werden die Fehler aus der Fehlerliste behoben. Dabei wird die Strategie zur Behebung des Fehlers dokumentiert, um Änderungen an den Dokumenten zu verdeutlichen. Dies geschieht mit Hilfe der Tabelle zur Dokumentation der Fehlerbehebungsstrategie im Dokument A-Verifikation.

Im Prozess Rework *System-Anforderungen* werden folgende Produkte modifiziert:

- A-Use Cases.

- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.

3.9 Prozess “Testfälle bearbeiten”

Entsprechend des für den Teilsystem-, System- und Akzeptanztest ausgewählten Testverfahrens werden Testpläne entwickelt. Ein mögliches Testkriterium zum Test der Funktionalität des Systems besteht in der Überdeckung der in den B-Anforderungen entwickelten Szenarien. Dafür werden beispielsweise in einem ersten Schritt Kombinationen von Äquivalenzklassen von Eingabewerten gebildet, die zu einem identischen funktionalen Verhalten führen. In einem zweiten Schritt werden die eigentlichen Testpläne erstellt, in denen für jede Eingangsvariable (Eingangswerte für die Testdurchführung) und jede Ausgangsvariable (Ausgangwerte, d. h. erwartetes Systemverhalten) konkrete Werte festgelegt werden.

Neben Testplänen zum Test des Verhaltens des Systems, sollten Testpläne zum Test der nicht-funktionalen Eigenschaften des Systems entwickelt werden.

Abbildung 46 zeigt den Prozess *Testfälle bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

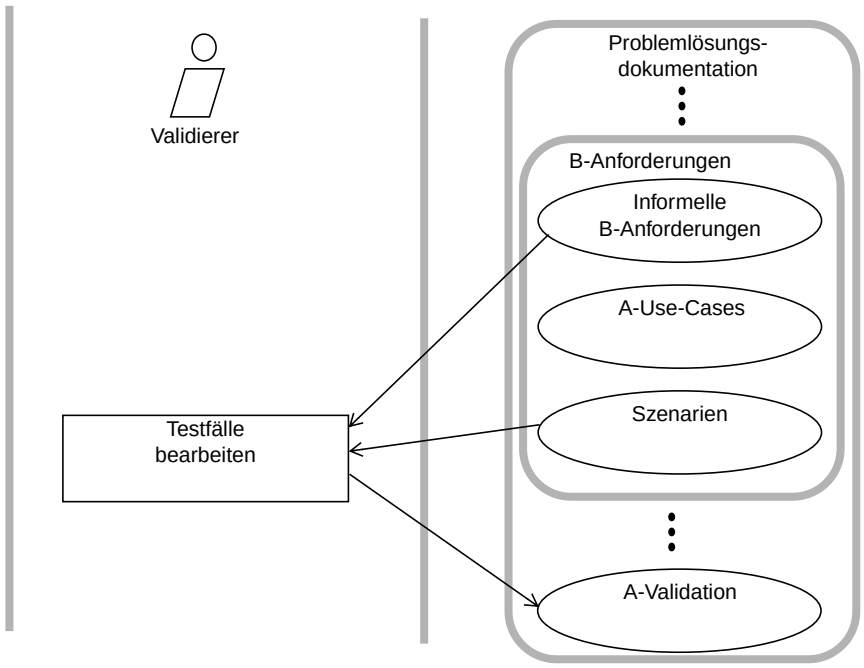


ABBILDUNG 46. Verwendete Produkte des Prozesses *Testfälle bearbeiten*

Folgende Produkte werden im Prozess *Testfälle bearbeiten* konsumiert:

- Nicht-funktionale B-Anforderungen (Teilprodukt der Informellen B-Anforderungen).
- Szenarien.

Folgendes Produkt wird produziert:

- A-Validation

3.10 Weiterführende Literatur

1. A. F. Ackerman, L. S. Buchwald, F. H. Lewski: *Software Inspections: An Effective Verification Process.* In: Donald J. Reifer (EDS.): *Software Management.* 5th Edition, IEEE Computer Society, Kapitel 10, S. 416-421, 1997.
2. Maher Awad, Juha Kuusela, Jürgen Ziegler: *Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion.* Prentice Hall, 1996.
3. Grady Booch, James Rumbaugh, Ivar Jacobson: *Das UML-Benutzerhandbuch.* Addison-Wesley Longman, 1999.
4. Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.
5. Alan Davis: *Software requirements: Analysis and Specification*. Prentice-Hall, 1990.
6. Richard A. DeMillo, W. Michael McCracken, R. J. Martin, John G. Passafiume: *Software testing and evaluation*. Benjamin/Cummings Publishing Inc., 1987.
7. Pankaj Jalote: *Integrated Approach to Software Engineering.* 2nd ed. Springer Verlag, 1997.
8. Barbara Kitchenham, Steve Linkman: *Validation, Verification, and Testing: Diversity Rules.* IEEE Software, Vol. 15, Nr. 4, S. 46-49, Juli 1998.
9. Antje von Knethen, Barbara Paech: *Reflections on the Object-Oriented Design of Embedded Systems.* OMER Workshop, May 28 - 29. 1999, P. Hofmann, A. Schürr (EDS.), OMER Workshop Proceedings Bericht Nr. 1999-01, Mai 1999.
10. P. Liggesmeyer, M. Rothfelder, M. Rettelbach, T. Ackermann: *Qualitätssicherung Software-basierter technischer Systeme - Problembereiche und Lösungsansätze.* GI Informatik Spektrum, Vol. 21, Nr. 5, S. 249-258 Oktober 1998.
11. Bernd Oesterreich: *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenburg-Verlag, 4., aktualisierte Auflage 1998.
12. James Rumbaugh, Michael Blaha, William Premerali: *Objektorientiertes Modellieren und Entwerfen.* Hanser Elektronik, München, 1993.
13. Wolfgang Wagenbichler: *Erstellung einer SW-Systemdokumentation mit StP.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 67653 Kaiserslautern, 1999. URL: http://wwwagse.informatik.uni-kl.de/access-system/publications.html#8
14. Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, Peter Haumer: *Scenarios in System Development: Current Practice.* IEEE Software, Vol. 15, Nr. 2, S. 34-45, März 1998.

15. Tsuneo Yamaura: *How To Design Practical Test Cases.* IEEE Software, Vol. 15, Nr. 6, S. 30-36, November 1998.

KAPITEL 4

System-Entwurf bearbeiten

Ziel des Prozesses *System-Entwurf bearbeiten* ist, ein realisierbares Software-System zu entwerfen, welches die System-Anforderungen aus der Problemlösungsdokumentation erfüllt. Darüber hinaus soll der Entwurf eine unabhängige Implementierung einzelner Komponenten (d. h. Klassen) erlauben. Um diese Ziele zu erreichen, werden die UML-Diagramme aus dem Prozess *System-Anforderungen bearbeiten* verfeinert und hinsichtlich ihrer Realisierbarkeit überarbeitet. Abbildung 47 gibt einen Überblick über die verwendeten Produkte im Prozess *System-Entwurf bearbeiten*.

Folgende Produkte werden konsumiert:

- *Namenskonventionen*. Dieses Dokument beinhaltet Konventionen für die Wahl von Bezeichnern in der Software-System-Dokumentation. Diese Konventionen erweitern die (auch in diesem Dokument enthaltenen) Konventionen für die Wahl von Bezeichnern in der Problemlösungsdokumentation.
- *Domänenwissen*. Dieses Dokument beschreibt typische Eigenschaften und Charakteristika der Anwendungsdomäne. Es umfasst insbesondere eine Beschreibung der Systemumgebung.

Folgende Produkte werden im Prozess *System-Entwurf bearbeiten* modifiziert:

- *Problemlösungsdokumentation*. Dieses Dokument beinhaltet die Anforderungen an das zu entwickelnde Software-System aus Sicht der Benutzer/Kunden und der Entwickler.
- *Software-System-Dokumentation.* Dieses Dokument umfasst eine Reihe von *Entwurfsentscheidungen*, die beispielsweise auf die Systemumgebung oder auf Qualitätsanforderungen an das Software-System zurückzuführen sind. Es beinhaltet den *UML-Entwurf* und die *Komponenten-Anforderungen*. Der *UML-Entwurf* beschreibt die endgültige Aufteilung des Software-Systems in Komponenten, die Interaktion zwischen diesen und das Verhalten der Komponenten. Die *Komponenten-Anforderungen* ergänzen die Beschreibungen der Komponenten aus dem *UML-Entwurf*. Die *Software-System-Dokumentation* umfasst außerdem eine *Verfolgbarkeitsmatrix-Entwurf*, mit deren Hilfe Zusammenhänge zwischen den System-Anforderungen und dem *UML-Entwurf* explizit gemacht werden. Des Weiteren enthält die Dokumentation *Dokumente zu Verifikationsaktivitäten* auf System-Entwurfsebene (z. B. eine Fehlerklassifikation, eine Checkliste zur Inspektion des Entwurfs, Fehlertabellen zur Dokumentation gefundener Fehler und Fehlerursachen) und *Dokumente zu Validationsaktivitäten* (z. B. Testpläne zum Test von Komponenten sowie Tabellen zur Dokumentation gefundener Fehlverhalten).

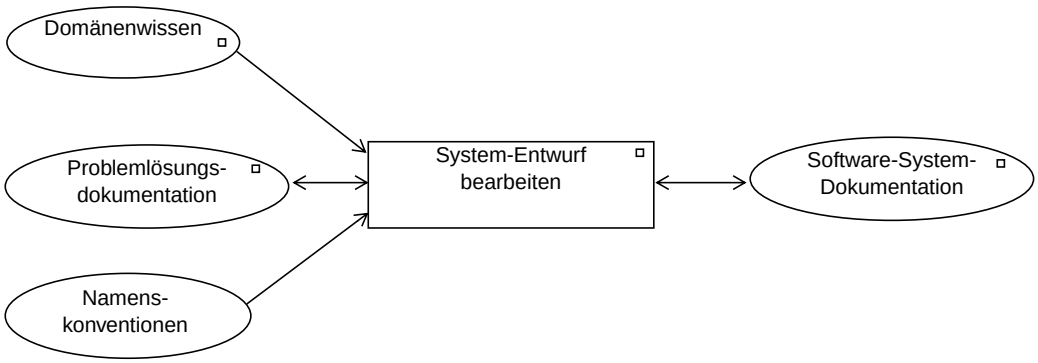


ABBILDUNG 47. Verwendete Produkte des Prozesses *System-Entwurf bearbeiten*

4.1 Prozessverfeinerung

Der Prozess *System-Entwurf bearbeiten* unterteilt sich in sechs Teilprozesse (siehe Abbildung 48). Zunächst werden aus den *B-Anforderungen* und dem *Domänenwissen Entwurfsentscheidungen* abgeleitet, die Festlegungen bezüglich der Implementierung des zu realisierenden Systems beschreiben. Ausgehend von den B- und E-Anforderungen werden im Prozess *UML-Entwurf bearbeiten* die UML-Diagramme verfeinert und hinsichtlich der *Entwurfsentscheidungen* überarbeitet. Insbesondere für die Verfeinerung der Diagramme sind *Namenskonventionen* wichtig, die eine einheitliche Namensgebung in den verschiedenen Entwicklungsprodukten und Teilsystemen gewährleisten. Ergebnis dieses Prozesses ist der *UML-Entwurf*, der die überarbeiteten UML-Diagramme enthält und Änderungen beschreibt. Im Prozess *Komponenten-Anforderungen erstellen* werden die Beschreibungen der Komponenten aus dem *UML-Entwurf* erweitert, sodass eine unabhängige Implementierung der Komponenten möglich ist. Im Prozess *Verfolgbarkeitsmatrix-Entwurf bearbeiten* wird eine *Verfolgbarkeitsmatrix-Entwurf* aufgestellt, in der Zusammenhänge zwischen den *E-Anforderungen* und dem *UML-Entwurf* beschrieben werden. Im Prozess *System-Entwurf verifizieren* werden der *UML-Entwurf*, die *Komponenten-Anforderungen* und die *Verfolgbarkeitsmatrix-Entwurf* gegen die *E-Anforderungen* und die *Entwurfsentscheidungen* inspiziert. Im Prozess *Komponenten-Testfälle bearbeiten* werden aus dem *UML-Entwurf* und den *Komponenten-Anforderungen* Testpläne für einen späteren Test der fertiggestellten Komponenten abgeleitet.

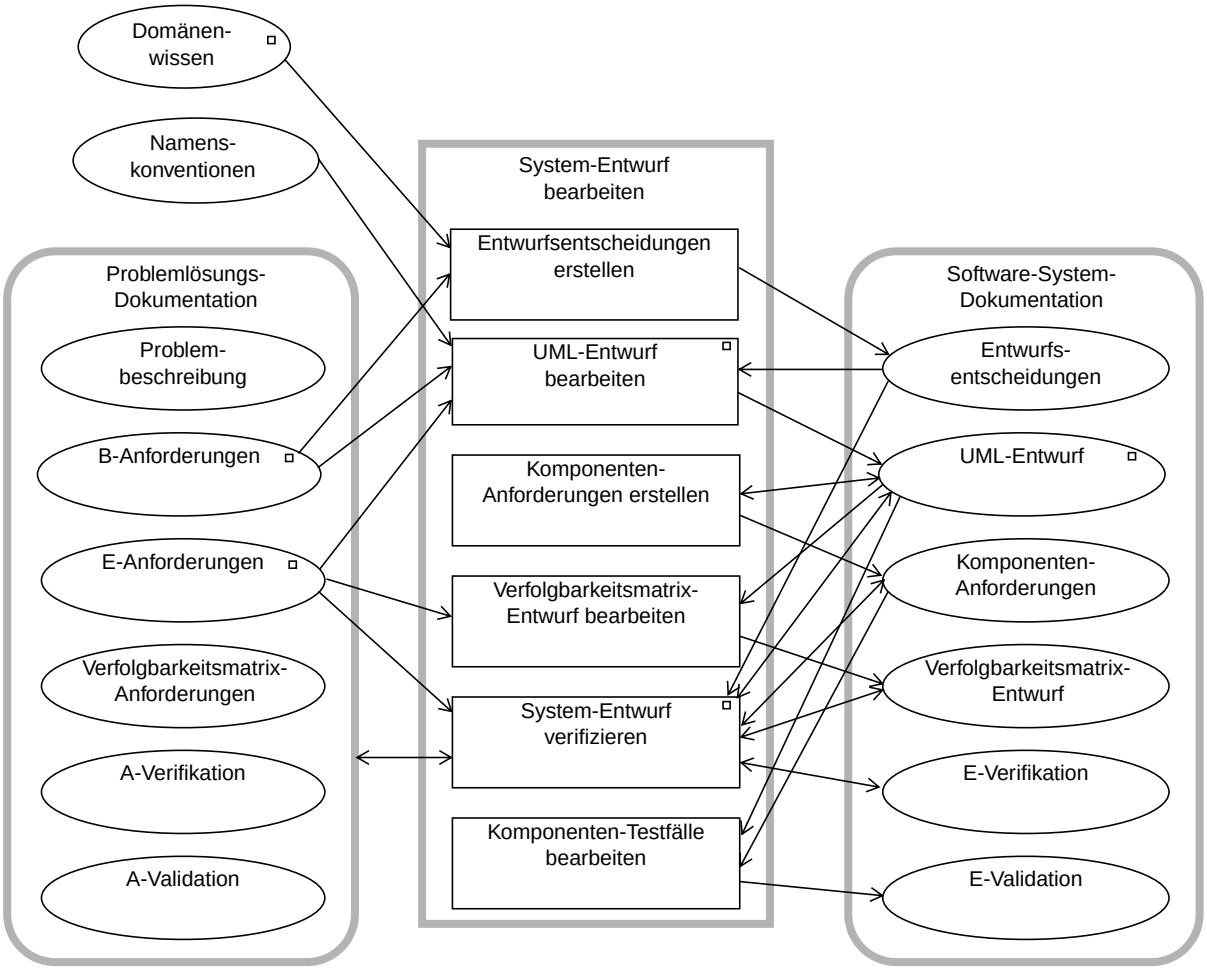


ABBILDUNG 48. Verfeinerung des Prozesses *System-Entwurf bearbeiten*

4.2 Produkte

Im Folgenden wird eine Übersicht über die im Prozess *System-Entwurf bearbeiten* verwendeten Produkte gegeben. Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte der Dokumente, soweit dies nicht bereits im vorangegangenen Kapitel erfolgt ist.

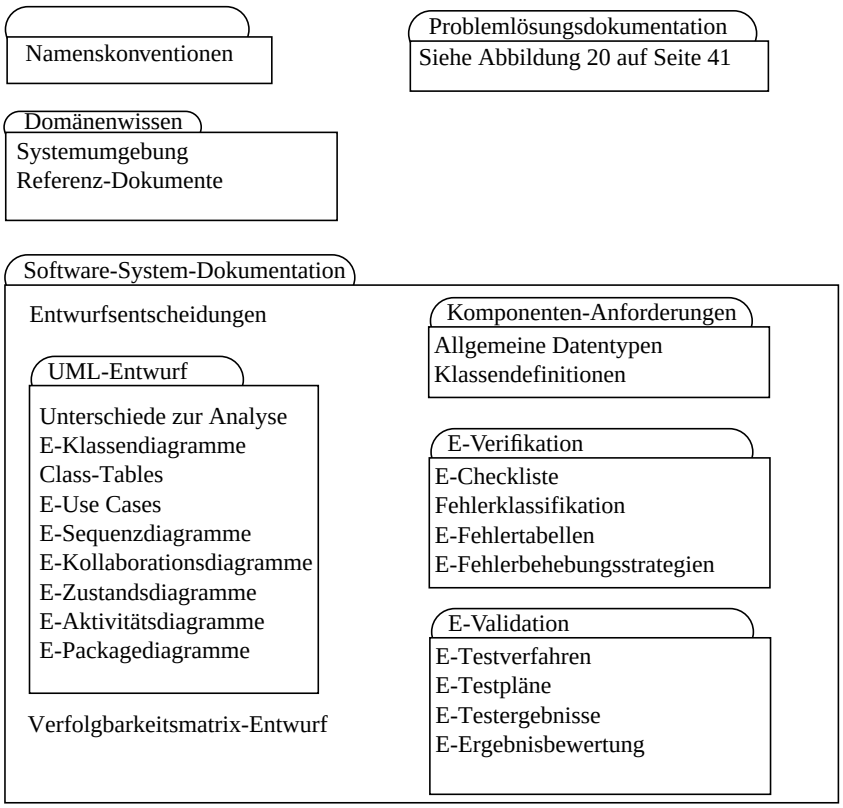


ABBILDUNG 49. Produktübersicht für den Prozess *System-Entwurf bearbeiten*

Entwurfsentscheidungen

Die *Entwurfsentscheidungen* beschreiben Implementierungsentscheidungen für das zu entwickelnde Software-System. Sie schränken die Menge der möglichen Realisierungen ein. *Entwurfsentscheidungen* resultieren beispielsweise aus der Umgebung, in der das Software-System eingesetzt werden soll (z. B. durch das verwendete Betriebssystem) oder aus Qualitätsanforderungen an das zu entwickelnde Software-System (z. B. Effizienz oder Portabilität). Die Wiederverwendung von Komponenten kann ebenfalls zu *Entwurfsentscheidungen* führen.

Die Abbildung 50 zeigt *Entwurfsentscheidungen* für das Gebäudeautomationssystem, die aus der Umgebung und aus der Wiederverwendung von Komponenten resultieren.

Beispiel *Entwurfsentscheidungen*

Die *Entwurfsentscheidungen* sind zur leichteren Referenzierung durchnummeriert. Die Entscheidungen EE1 und EE2 lassen sich auf Eigenschaften der Systemumgebung zurückführen. EE3 ergibt sich aus der Wiederverwendung einer Komponente.

EE	Beschreibung der Entwurfsentscheidung
EE1	Das Zielsystem ist ein zentraler Steuerungsrechner mit einem Portbaustein, über den • alle Sensoren abgefragt • alle Aktuatoren angesteuert werden können.
EE2	Parallelität wird durch Verwendung eines einfachen Schedulers simuliert.
EE3	Die Zeit wird durch eine wiederverwendbare Komponente Timer realisiert.

ABBILDUNG 50. Ausschnitt aus den Entwurfsentscheidungen für ein Gebäudeautomationssystem

Neben diesen *Entwurfsentscheidungen* gibt es noch eine Reihe allgemeiner *Entwurfskriterien*, die bei der Überarbeitung und Verfeinerung des UML-Entwurfs

eine Rolle spielen. Es lassen sich Optimierungs- und Verfeinerungskriterien unterscheiden:

Optimierungskriterien

Auf folgende Aspekte ist zu achten:

1. Gemeinsamkeiten zwischen Klassen werden in Oberklassen zusammengefasst.
2. Klassen ohne Funktionalität werden entfernt.
3. Die Anwendbarkeit von Entwurfsmustern (“Design Pattern”) wird geprüft.
4. Die Kopplung im System wird reduziert. Dafür muss geprüft werden, ob sich Beziehungen zwischen Klassen oder Interaktionen zwischen Instanzen von Klassen vereinfachen lassen.

Verfeinerungskriterien

Auf folgende Aspekte ist zu achten:

1. Im Klassendiagramm werden Multiplizitäten für die Klassen, die miteinander in Beziehung stehen, angegeben.
2. Richtungen für Assoziationen werden festgelegt.
3. Ziel und Quelle von Werten (z. B. Bereitstellung eines Werts von der Klasse ControlPanelFM für eine Klasse, die am Raum aggregiert ist) müssen geeignet verbunden sein. Wichtig ist, dass Werte eine Klasse, wenn diese sie benötigt, wirklich erreichen. Die Verbindung der Klassen kann auf verschiedene Arten hergestellt werden, beispielsweise durch:
   - Assoziationen zwischen zwei Klassen, die bei der Initialisierung hergestellt werden,
   - definierte Durchreichmethoden, die einen Wert weiterreichen,
   - das Bereitstellen einer getObject-Methode von der Klasse, die die Objekte verwaltet, die den Wert verarbeiten oder
   - durch die Anwendung des Singleton-Pattern, wenn es von der Klasse, die den Wert bereitstellt, nur eine Instanz gibt.

   Vom einzelnen Fall hängt ab, welche Lösung am geeignetsten ist.
4. Die Parameterliste jeder Methode muss vervollständigt werden.
5. Es müssen Datentypen von Parametern und Rückgabewerten angegeben werden.
6. Die Sichtbarkeit von Methoden und Attributen (public, protected, private) wird festgelegt.

7. Defaultwerte für Attribute werden, wenn möglich, festgelegt.
8. Eine Klasse wird als abstrakt definiert, wenn keine Instanzen von ihr abgeleitet werden.
9. Konstruktoren und Destruktoren für die Initialisierung und das Löschen von Instanzen von Klassen müssen angelegt werden. Dafür ist eine Initialisierungsreihenfolge der Klassen zu definieren.
10. Wird in einem Zustandsdiagramm eine Methode einer anderen Klasse aufgerufen: Beim Methodenaufruf wird der Klassename mit angegeben: Klasse.Methode()

UML-Entwurf

Der *UML-Entwurf* beschreibt die Zerlegung des zu entwickelnden Software-Systems in Komponenten, die implementiert werden sollen. Außerdem wird die Interaktion und das Verhalten der Komponenten beschrieben. Dafür werden die UML-Diagramme aus den E-Anforderungen hinsichtlich der Entwurfsentscheidungen und der weiteren Entwurfsaspekte überarbeitet und verfeinert. Unterschiede zwischen den UML-Diagrammen aus den E-Anforderungen und denen des *UML-Entwurfs* werden im Dokument *Unterschiede zur Analyse* festgehalten. Der *UML-Entwurf* umfasst folgende UML-Diagramme: 1. *Packagediagramme*, 2. *Klassendiagramme*, 3. Klassenbeschreibungen *(Class-Tables)*, 4. ein Use Case-Diagramm (*E-Use Cases*)[1], 5. *Sequenzdiagramme*, 6. *Kollaborationsdiagramme*, 7. *Zustandsdiagramme* und 8. *Aktivitätsdiagramme*.

Unterschiede zur Analyse

Im Dokument *Unterschiede zur Analyse* werden Unterschiede zwischen den A-Klassendiagrammen aus den E-Anforderungen und den E-Klassendiagrammen aus dem UML-Entwurf dokumentiert. Dabei werden Änderungen für jede Klasse aus dem UML-Entwurf aufgelistet.

Mit Hilfe des Dokuments ist es möglich, die Auswirkungen der Entwurfsentscheidungen auf die A-Klassendiagramme für jede Klasse nachzuvollziehen. Die Struktur der E-Klassendiagramme (Klassen und deren Beziehungen) wird mit diesem

1. Da Use Cases auch auf Anforderungsebene auftreten können, werden die Use Cases auf Entwurfsebene im Folgenden auch als *E-Use Cases* bezeichnet. Dies gilt analog auch für weitere Bezeichner mit dem Präfix “E-”.

Dokument begründet. Das Dokument erleichtert einerseits eine Überprüfung der Konsistenz zwischen den E- und den A-Klassendiagrammen, da Unterschiede begründet werden müssen. Andererseits unterstützt es eine Propagierung von Änderungen an den A-Klassendiagrammen zu den E-Klassendiagrammen.

Das Beispiel (siehe Abbildung 51) zeigt dokumentierte Änderungen an den Klassen “LightControlUnit” und “LightController” aus dem Teilsystem “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *Unterschiede zur Analyse*

Die kurze Beschreibung erläutert für jede Klasse aus dem *UML-Entwurf*, wie sich die *Entwurfsentscheidungen* und *Optimierungskriterien* auswirken. Aktive Klassen werden im betrachteten Gebäudeautomationssystem beispielsweise von einem Scheduler abgeleitet, der für die regelmäßige Aktivierung der Klassen sorgt.

Klasse LightControlUnit

Die Methoden personIn() und personOut() wurden zur Methode occupancyMessage(zOccupancyType) zusammengefasst. Die Methoden fmSwitchOn() und fmSwitchOff() wurden zur Methode fmLightControl(zLightStatusType) zusammengefasst.

Klasse LightController

Die Klasse LightController wird aufgrund ihres aktiven Verhaltens von der Klasse ScheduledFct abgeleitet. Die Attribute sectionNr und roomNr wurden eingefügt, um Anwesenheitsdetektoren und Lichter auf der Hardware-Ebene eindeutig initialisieren bzw. ansprechen zu können. Die Methoden switchLightOn() und switchLightOff() wurden in der Methode lightControl(zLightStatusType) zusammengefaßt. Die Methode setLightControlUnit() dient der Initialisierung.

ABBILDUNG 51. Dokumentation von Änderungen am A-Klassendiagramm

E-Packagediagramme

Packages werden verwendet, um die Aufteilung des Gesamtsystems in Teilsysteme (*Packages*) aufzuzeigen und Beziehungen zwischen den Teilsystemen zu verdeutlichen. Im Vergleich zu den A-Packagediagrammen kann es im Entwurf weitere Teilsysteme geben, die aus den Entwurfsentscheidungen resultieren und sich beispielsweise mit Implementierungsdetails beschäftigen (z. B. mit der Realisierung der Zeit).

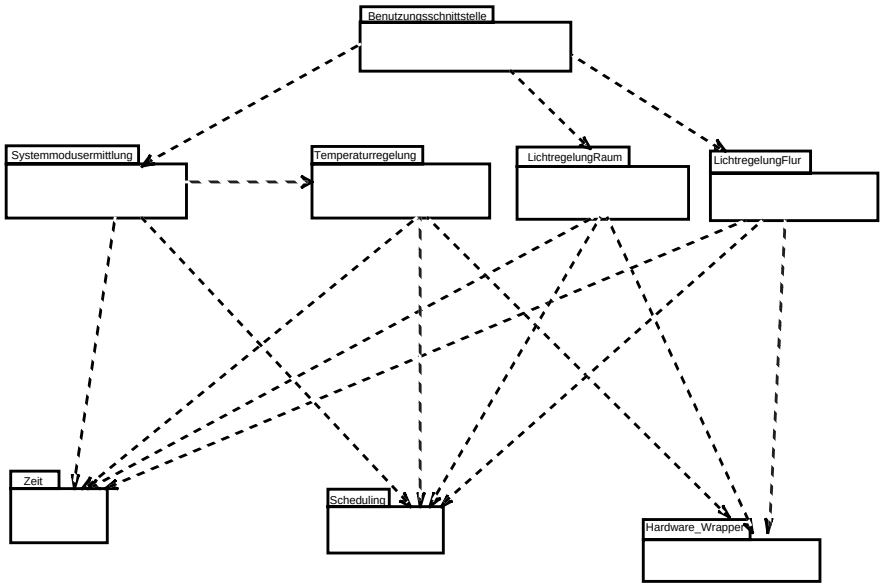
Jedes Package wird mit einer kurzen Beschreibung erläutert.

Abbildung 38 zeigt ein *E-Packagediagramm* für das Gebäudeautomationssystem.

Beispiel *E-Packagediagramm*

Zu den sechs Teilsystemen aus den E-Anforderungen (siehe Abbildung 23, „Teilsysteme eines Gebäudeautomationssystems und Erläuterung des Package “LichtregelungFlur”,“ auf Seite 48) kommen zwei weitere Teilsysteme (Zeit und Scheduling) hinzu. Das Teilsystem “Zeit” umfasst alle Klassen, die an der Realisierung der Zeitfunktionalität beteiligt sind. Das Teilsystem “Scheduling” umfasst alle Klassen, die an der Realisierung des Schedulingmechanismusses beteiligt sind.

.



Package: LichtregelungFlur
Dieses Package enthält Klassen, die die Regelung des Lichts in den Flurabschnitten übernehmen. Werte, die dafür benötigt werden, werden mit Hilfe der Klassen des Packages Benutzungsschnittstelle gesetzt.

ABBILDUNG 52. E-Packagediagramm für ein Gebäudeautomationssystem

E-Klassendiagramme

Pro Teilsystem existiert im UML-Entwurf ein *Klassendiagramm*, welches die Beziehungen zwischen den *Klassen* sowie Attribute und Methoden von *Klassen* beschreibt.

In jedem *Klassendiagramm* sind die Methoden und Attribute einer *Klasse* dargestellt, die für die Erbringung der im *Klassendiagramm* beschriebenen Funktionalität erforderlich sind. Tritt eine *Klasse* in mehreren *Klassendiagrammen* auf (d. h., wird sie zur Erbringung unterschiedlicher Funktionalitäten benötigt), dann zeigen einzelne *Klassendiagramme* nur Sichten auf diese *Klasse*.

Die *E-Klassendiagramme* sind verfeinerte und überarbeitete A-Klassendiagramme. Da das *E-Klassendiagramm* die endgültige Struktur des Software-Systems beschreibt und Beziehungen zwischen den *Klassen* entsprechend des Diagramms

realisiert werden, wird das A-Klassendiagramm unter Entwurfsgesichtspunkten überarbeitet und verfeinert. Parameter und Rückgabewerte werden bestimmt und Datentypen festgelegt. Damit Werte im Software-System *Klassen* erreichen können, müssen Durchreichmethoden in *Klassen* ergänzt werden, das Initialisieren und Löschen von Objekten muss sichergestellt werden und die System-Struktur sollte optimiert werden, d. h. die Kopplung im Software-System wird minimiert und die Kohäsion maximiert. Außerdem wird das A-Klassendiagramm hinsichtlich der getroffenen Entwurfsentscheidungen (z. B. hinsichtlich Verteilung oder Parallelität) überarbeitet.

Jedes Klassendiagramm wird kurz erläutert.

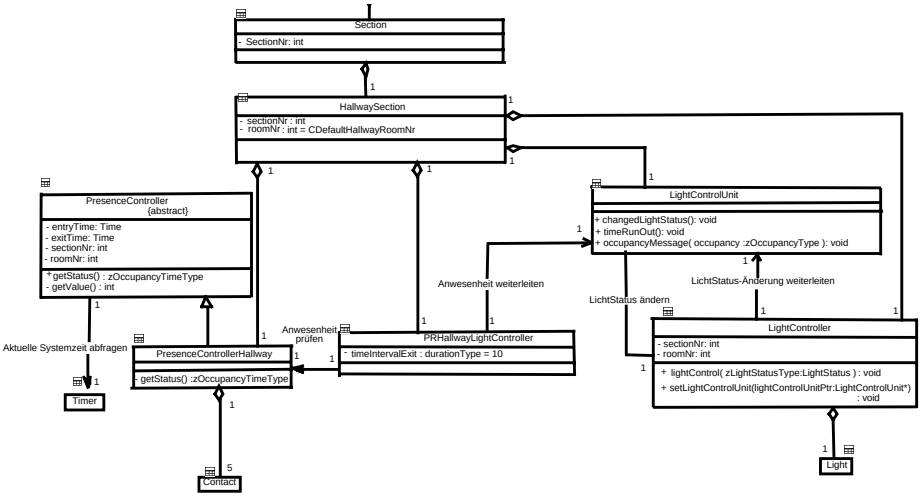
Abbildung 53 zeigt einen Ausschnitt aus dem *E-Klassendiagramm* des Teilsystems “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *E-Klassendiagramm*

Das *E-Klassendiagramm* des Teilsystems “LichtRegelungFlur” ist ein verfeinertes A-Klassendiagramm (vgl. Abbildung 24, „Klassendiagramm des Teilsystems LichtregelungFlur,“ auf Seite 50). Aus dem *E-Klassendiagramm* können im Gegensatz zum A-Klassendiagramm Parameter und Rückgabewerte sowie deren Datentypen entnommen werden. Außerdem sind Defaultwerte für Attribute im *E-Klassendiagramm* angegeben. Beispielsweise ist das Attribut “timeIntervalExit” der Klasse “PRHallwayLightController” vom Typ “DurationType” und hat einen Defaultwert von “10 min”.

Die Ableitung der aktiven *Klasse* “LightController” von der Klasse “ScheduledFct” kann nicht diesem *Klassendiagramm* entnommen werden. Die Vererbungsstruktur wird im *Klassendiagramm* “Scheduling” deutlich, das Klassen und Beziehungen zur Realisierung des Scheduling-Mechanismusses umfasst. Attribute und Methoden der Sensor- und Aktuatorklassen sind im *Klassendiagramm* “Hardware-Wrapper” beschrieben. Ebenfalls wurden die Klassen “Section” und HallwaySection” um Durchreichmethoden (z. B. setTimeIntervalExit (timeIntervalExitParam:durationType)) erweitert. Dies kann dem Klassendiagramm Benutzungsschnittstelle entnommen werden.

.



Klassendiagramm LichtregelungFlur
Dieses Diagramm zeigt alle Klassen, die an der Erbringung der Funktionalität der Lichtregelung in den Flurabschnitten beteiligt sind.

ABBILDUNG 53. Ausschnitt aus dem E-Klassendiagramm für das Teilsystem “LichtRegelungFlur”

Klassenbeschreibungen (Class-Tables)

Die *Klassenbeschreibungen* (*Class-Tables*) erläutern die Attribute und öffentlichen Methoden jeder *Klasse* im Software-System informell. Im Gegensatz zu den Klassendiagrammen beschreiben die *Klassenbeschreibungen* keine Sichten auf eine *Klasse*, sondern fassen die Informationen bez. einer *Klasse* aus allen Klassendiagrammen zusammen. Die *Klassenbescheibungen* sind vergleichbar mit dem Data-Dictionary der E-Anforderungen. Allerdings werden Attribute und Methoden der *Klassen* in den *Klassenbeschreibungen* detaillierter beschrieben.

Abbildung 54 zeigt einen Ausschnitt aus der *Klassenbeschreibung* der *Klasse* “LightController” des Teilsystems “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *Klassenbeschreibung*

LightController		
Attribute	**Type**	**Default Value**
roomNr	int	
Operation	**Arguments**	**Return Type**
lightControl	zLightStatusType:LightStatus	void
setLightControlUnit	lightControlUnitPtr:LightControlUnit*	void

Attribute der Klasse LightController:

Attribut	Erläuterung
roomNr	Das Attribut roomNr wird zur Erzeugung eines Objekts der Klasse Light benötigt, das die Flurlichtanlage des Flurabschnitts repräsentiert. Da sich die Lichtanlage im Flurabschnitt befindet und nicht in einem Raum, wird roomNr mit einem Wert initialisiert, der diesen Sachverhalt kennzeichnet. Die Lichtanlage kann eindeutig durch ihren Namen identifiziert werden.

Methodenschnittstellen der Klasse LightController:

Signatur: **void lightControl(zLightStatusType:LightStatus)**

Beschreibung: Die Methode requestLightStatus() gibt den Status der Lichtanlage (Licht ein-/ausgeschaltet) zurück.

Signatur: **void setLightControlUnit(lightControlUnitPtr:LightControlUnit*)**

Beschreibung: Die Methode setLightControlUnit() dient der nachträglichen Übergabe eines Pointers auf ein Objekt der Klasse LightControlUnit. Die Notwendigkeit dieser Methode ergibt sich aus der Initialisierungsreihenfolge (das LightController-Objekt wird vor dem LightControlUnit-Objekt erzeugt) der Objekte der Klassen LightControlUnit und LightController.

ABBILDUNG 54. Ausschnitt aus der Klassenbeschreibung der Klasse “LightController”

E-Use Case-Diagramm (E-Use Cases)

Zur Einordnung der verfeinerten *E-Sequenzdiagramme* wird das *Use Case-Diagramm* (*A-Use Cases*) aus den *B-Anforderungen* wiederholt. Das Diagramm und eine Erläuterung kann Abbildung 26, „Ausschnitt aus dem Sequenzdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit,“ auf Seite 52 entnommen werden. An den A-Use Cases werden keine Veränderungen vorgenommen.

E-Sequenzdiagramme

Die *E-Sequenzdiagramme* des UML-Entwurfs beschreiben das Verhalten des Software-Systems durch die Interaktion zwischen den zu implementierenden Objekten. Die *E-Sequenzdiagramme* sind verfeinerte und überarbeitete A-Sequenzdiagramme. Sie sind an die Entwurfsentscheidungen und -kriterien angepasst.

Abbildung 10 zeigt ein *E-Sequenzdiagramm* zu “Szenario1” des Use Cases “Flurlichtregelung_durch_Anwesenheit”.

Beispiel *E-Sequenzdiagramme*

Das *E-Sequenzdiagramm* verfeinert das A-Sequenzdiagramm zu Szenario 1 (Abbildung 26, „Ausschnitt aus dem Sequenzdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit,“ auf Seite 52). Die Nachrichten, die zwischen den Objekten verschickt werden, sind im *E-Sequenzdiagramm* beispielsweise angereichert um Parameter. Außerdem wurden Entscheidungen hinsichtlich der Klassenstruktur umgesetzt, wie das Zusammenfassen von Methoden (z. B. personIn() und personOut() zu occupancyMessage(zOccupancyType)).

Existierende CASE-Werkzeuge (z. B. StP/UML) erlauben automatische Überprüfungen der Konsistenz zwischen verschiedenen UML-Diagrammen, beispielsweise zwischen Sequenzdiagrammen und Klassendiagrammen. Bei StP/UML wird z. B. überprüft, ob die Nachrichten mit Parametern, die ein Objekt im Sequenzdiagramm geschickt bekommt, auch im Klassendiagramm als Methoden der Klasse definiert sind. Da ein Sequenzdiagramm eine konkrete Benutzung des Software-Systems wiederspiegelt, müssten im Sequenzdiagramm Parameter eigentlich konkrete Werte annehmen (z. B. lightControl(zLightStatusType = cLightOn)). Eine konkrete Belegung der Parameter mit Werten macht allerdings die Überprüfung der Konsistenz zwischen Sequenzdiagramm und Klassendiagramm für StP/UML unmöglich. Aus diesem Grund werden im Beispiel konkrete Werte von Parametern und Rückgabewerte mit Bedingungen formuliert.

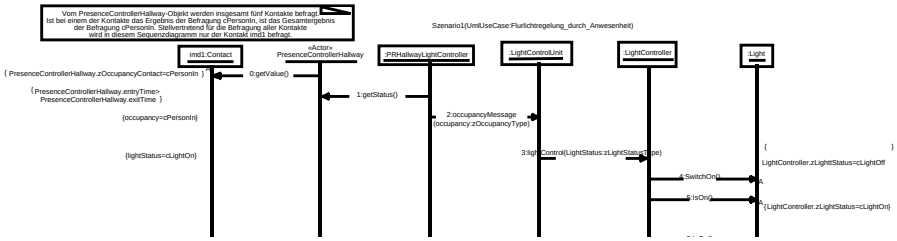


ABBILDUNG 55. Sequenzdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit

E-Kollaborationsdiagramme

Die *E-Kollaborationsdiagramme* des UML-Entwurfs beschreiben das Verhalten des Software-Systems durch die Interaktion zwischen zu implementierenden Objekten. Die *E-Kollaborationsdiagramme* verfeinern die A-Kollaborationsdiagramme aus den E-Anforderungen. Sie sind an die Entwurfsentscheidungen angepasst und hinsichtlich der Entwurfsaspekte überarbeitet worden.

Unter Verwendung eines CASE-Werkzeugs (z. B. StP/UML) lassen sich die *E-Kollaborationsdiagramme* automatisch aus den E-Sequenzdiagrammen generieren.

Abbildung 56 zeigt ein *Kollaborationsdiagramm* zu “Szenario1” des Use Cases “Flurlichtregelung_durch_Anwesenheit”.

Beispiel *E-Kollaborationsdiagramm*

Das E-Kollaborationsdiagramm verfeinert das A-Kollaborationsdiagramm (Abbildung 27, „Kollaborationsdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit,“ auf Seite 53) und wurde aus dem E-Sequenzdiagramm generiert. Bei der Generierung mit StP/UML werden allerdings Parameter und Datentypen nicht übernommen.

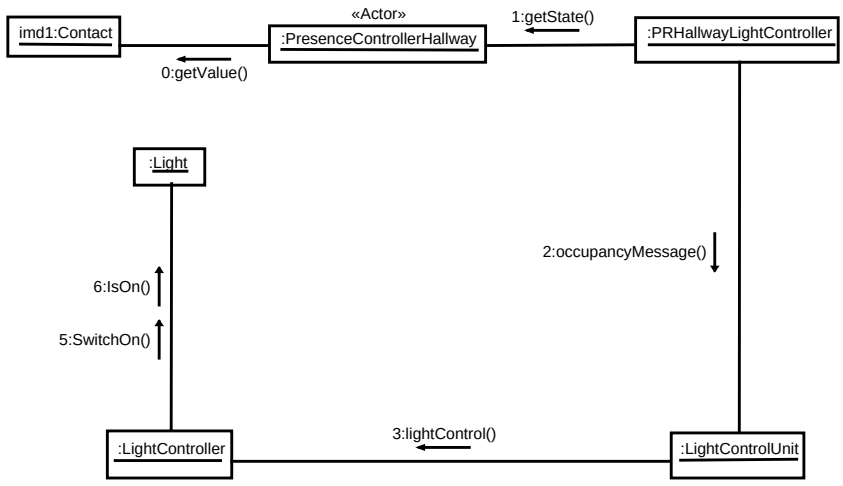


ABBILDUNG 56. E-Kollaborationsdiagramm zu Szenario1 - Flurlichtregelung durch Anwesenheit

E-Zustandsdiagramme

Das dynamische Verhalten einer Klasse wird mit Hilfe eines *Zustandsdiagramms* beschrieben. Die *E-Zustandsdiagramme* aus dem UML-Entwurf verfeinern die A-Zustandsdiagramme. Sie sind an die Entwurfsentscheidungen angepasst und hinsichtlich der Entwurfsaspekte überarbeitet worden.

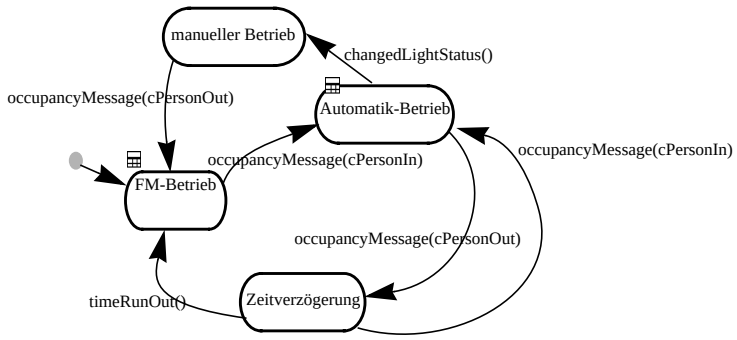
Jedes Zustandsdiagramm wird informell beschrieben. Dabei sollten insbesondere die verschiedenen Zustände erläutert werden.

Abbildung 57 zeigt ein *E-Zustandsdiagramm* für die Klasse “LightControlUnit” des Teilsystems “LichtRegelungFlur” eines Gebäudeautomationssystems.

Beispiel *E-Zustandsdiagramme*

Das *E-Zustandsdiagramm* verfeinert das A-Zustandsdiagramm für die Klasse “LightControlUnit” (Abbildung 28, „Zustandsdiagramm der Klasse

LightControlUnit mit einem Ausschnitt aus den vorhandenen Entry-, Do- und Exit-Aktionen," auf Seite 55). Die Nachrichten, die ein Objekt der Klasse verschickt oder geschickt bekommt, sind erweitert um Parameter und Rückgabewerte. Außerdem wurden Entscheidungen hinsichtlich der Klassenstruktur berücksichtigt, wie das Zusammenfassen von Methoden.



Entry-Aktion für State: Automatik-Betrieb

LightController.lightControl(cLightOn)

Entry-Aktion für State: FM-Betrieb

LightController.lightControl(cLightOff)

Internal-Aktion für State: FM-Betrieb

Event: fmLightControl(cLightOn) Action: LightController.lightControl(cLightOn)
Event: fmLightControl(cLightOff) Action: LightController.lightControl(cLightOff)

Zustandsdiagramm der Klasse LightControlUnit:
Die Klasse kennt vier Zustände FM-Betrieb, Automatik-Betrieb, manueller Betrieb und Zeitverzögerung. Während des FM-Betriebs ist es dem Hausmeister erlaubt (niemand ist im Flur anwesend) das Flurlicht über das ControlPanelFM einzuschalten. Wenn ein Benutzer im Flurabschnitt anwesend ist, wird unterschieden, ob er den Flurlichtschalter betätigt hat (manueller Betrieb) oder nicht (Automatik-Betrieb). Der manuelle Betrieb wird wieder aufgehoben, wenn alle Benutzer den Flurabschnitt verlassen haben. Im Zustand Zeitverzögerung wird verweilt, bis alle Benutzer über eine bestimmte Zeitspanne den Flur verlassen haben. Erst dann wird das Licht ausgeschaltet.

ABBILDUNG 57. Zustandsdiagramm der Klasse LightControlUnit mit einem Ausschnitt aus den vorhandenen Entry-, Do- und Exit-Aktionen

E-Aktivitätsdiagramme

Methoden von Klassen führen teilweise komplexe Operationen aus. Die *E-Aktivitätsdiagramme* erlauben deren Zerlegung. Einige komplexe Operationen wurden bereits in den E-Anforderungen identifiziert. Die existierenden A-Aktivitätsdia-

gramme aus den E-Anforderungen werden zu *E-Aktivitätsdiagrammen* verfeinert. Außerdem werden *Aktivitätsdiagramme* für weitere komplexe Operationen aus dem UML-Entwurf ergänzt. Wenn die Operationen in den E-Anforderungen nur informell beschrieben wurden, dann ist es sinnvoll, diese nun mit Hilfe von Pseudo-Code oder Aktivitätsdiagrammen zu präzisieren. Auf diese Weise werden Fragen frühzeitig geklärt.

Abbildung 58 zeigt ein *E-Aktivitätsdiagramm* für die Methode “check()” der Klasse “TimeIntervalsExit” des Teilsystems Systemmodusermittlung für das Gebäudeautomationssystem.

Beispiel *E-Aktivitätsdiagramm*

Die Methode check() der Klasse TimeIntervalsExit überprüft, ob ein vom Benutzer eingegebener Wert innerhalb eines Akzeptanzintervalls liegt. Wenn dies der Fall ist, dann wird der eingegebene Wert als aktueller Wert verwendet. Wenn der Wert des Benutzers nicht innerhalb des Akzeptanzintervalls liegt, wird überprüft, ob der Wert über der oberen Grenze oder unter der unteren Grenze des Intervalls liegt. Je nachdem wird die obere oder untere Grenze als neuer Wert verwendet.

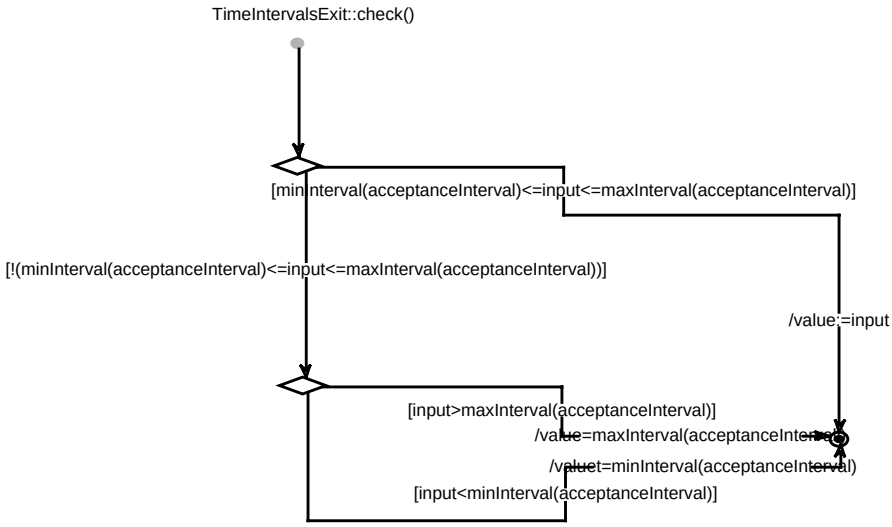


ABBILDUNG 58. E-Aktivitätsdiagramm zur Methode check() der Klasse TimeIntervalExit

Verfolgbarkeitsmatrix-Entwurf

Die *Verfolgbarkeitsmatrix-Entwurf* zeigt Beziehungen zwischen den E-Anforderungen und dem UML-Entwurf auf. Attribute und Methoden aus den E-Anforderungen und Attribute und Methoden aus dem UML-Entwurf werden für jede Klasse des Entwurfs gegenübergestellt. Ein Kreuz in der Tabelle zeigt, dass ein Attribut/Methode des UML-Entwurfs an der Realisierung eines Attributs/Methode der E-Anforderungen beteiligt ist. Die *Verfolgbarkeitsmatrix* unterstützt eine Überprüfung der Konsistenz zwischen den E-Anforderungen und dem UML-Entwurf und die Durchführung später evtl. auftretender Änderungen.

Abbildung 59 zeigt einen Ausschnitt aus einer *Verfolgbarkeitsmatrix-Entwurf* für die Klasse “LightController” des Teilsystems “LichtRegelungFlur”.

Beispiel *Verfolgbarkeitsmatrix-Entwurf*

Die Zeilen geben die Attribute und Methoden der Klasse “LightController” aus den E-Anforderungen wieder, die Spalten die Attribute und Methoden der Klasse aus dem UML-Entwurf. Fehlende Kreuze in Spalten oder Zeilen sollten im Dokument *Unterschiede zur Analyse* erklärt sein.

		LightController Entwurf				
		lightStatus	sectinNr	roomNr	lightControl()	setLightControlUnit()
LightController E-Anforderungen	lightStatus	X				
	switchLightOn()				X	
	switchLightOff()				X	

ABBILDUNG 59. Ausschnitt aus der Verfolgbarkeitsmatrix der Klasse LightController

Komponenten-Anforderungen

Die *Komponenten-Anforderungen* erweitern die Beschreibung der Komponenten (d. h. Klassen) aus dem UML-Entwurf. Hauptziel der *Komponenten-Anforderungen* ist es, eine unabhängige Implementierung der Komponenten zu erlauben. Dafür werden verwendete Datentypen definiert. Private Methoden von Klassen sind bereits im E-Klassendiagramm aufgetreten, allerdings noch nicht beschrieben worden. Konstruktoren und Destruktoren von Klassen werden aufgelistet. Außerdem wird angegeben, wie Assoziationen und Aggregationen zu realisieren sind. Von einer Klasse importierte Datentypen und Methoden werden beschrieben.

Die Komponenten-Anforderungen umfassen zwei Kapitel: 1. *Allgemeine Datentypen* und 2. *Klassendefinitionen*.

Allgemeine Datentypen

Die *Allgemeinen Datentypen* definieren die Datentypen, die im UML-Entwurf verwendet wurden. Die Datentypen können wiederverwendet oder neu definiert sein.

Abbildung 60 zeigt einen Ausschnitt aus den definierten Datentypen des UML-Entwurfs für das Gebäudeautomationssystem.

Beispiel *Allgemeine Datentypen*

Bei jedem Datentyp wird angegeben, ob er wiederverwendet oder neu implementiert wird. Die angegebene Definition nimmt bereits Bezug auf die konkrete Implementierungssprache C++.

C++ Typdefinition	Art	Beschreibung
typedef unsigned int timeCompType	Wiederverw.	Beschreibt einen Bestandteil der Zeit (Stunden, Minuten oder Sekunden)
durationType timeCompType	Wiederverw.	Beschreibt eine Zeitdauer in Minuten.
typedef enum {cPersonIn, cPersonOut} zOccupancyType	Neu	Mögliche Werte für den Anwesenheitstatus.
typedef enum {cLightOff, cLightOn} zLightStatusType	Neu	Mögliche Werte für die Flurlichtanlage.

ABBILDUNG 60. Ausschnitt aus den definierten Datentypen eines Gebäudeautomationssystem

Klassendefinitionen

Die *Klassendefinitionen* beschreiben jeweils für eine Klasse, wie Assoziationen und Aggregationen zu anderen Klassen realisiert werden. Darüber hinaus werden Konstruktoren, Destruktoren, private Methoden, sowie importierte Datentypen und Methoden anderer Klassen beschrieben. Zusammen mit den Klassenbeschreibun-

gen und den Zustands- und Aktivitätsdiagrammen des UML-Entwurfs erlauben die *Klassendefinitionen* eine unabhängige Implementierung einzelner Klassen.

Die Liste der importierten Methoden kann aus den Informationen der E-Sequenzdiagramme des UML-Entwurfs heraus mit Hilfe eines CASE-Werzeugs (z. B. StP/UML) generiert werden.

Abbildung 61 zeigt einen Ausschnitt aus einer *Klassendefinition* der Klasse “Light-Controller”.

Beispiel *Klassendefinition*

Assoziationen werden im Beispiel nur in die Richtung, in die eine Klasse Methoden einer anderen Klasse aufruft, realisiert (durch einen Zeiger auf die Klasse). Aggregationen von anderen Klassen werden ebenfalls als Zeiger auf diese Klassen realisiert.

Klasse LightController

Assoziationen zu Objekten anderer Klassen

Keine

Aggregationen von Objekten anderer Klassen

Pointername	Klassenname des aggregierten Objektes
lightPtr	Light

Methodenschnittstellen:

Signatur: **LightController(sectionNr:int, initialRoomNr:int)**

Beschreibung: Konstruktor

Imports:

Importierte Methoden

Herkunft: Light

- IsOn
- SwitchOn
- SwitchOff

Herkunft: LightControlUnit

- changedLightStatus

Importierte Datentypen

Herkunft: SimpleTypes

- zLightStatusType

ABBILDUNG 61. Ausschnitt aus der Klassendefinition der Klasse LightController

E-Verifikation

Das Dokument *E-Verifikation* enthält Hinweise zu und Ergebnisse von durchgeführten Verifikationsaktivitäten auf Entwurfsebene.

Im Folgenden wird sich auf eine Inspektion mit Checklisten als Verifikationsaktivität eingeschränkt. Das Dokument E-Verifikation umfasst dann: 1. eine *Checkliste,*

2. eine *Fehlerklassifikation*, 3. eine *Liste der gefundenen Fehler* und 4. eine *Beschreibung der Fehlerbehebungsstrategien*.

E-Checkliste

Die *E-Checkliste* unterstützt das Lesen des zu verifizierenden UML-Entwurfs, der Komponenten-Anforderungen und der Verfolgbarkeitsmatrix-Entwurf und erleichtert das Finden von Problemen in den Dokumenten.

Pro Dokument müssen - wie bei der Verifikation der System-Anforderungen - drei Aspekte überprüft werden: 1. Sind die Dokumentationsrichtlinien erfüllt (d. h. ist das Dokument syntaktisch korrekt), 2. Ist die Qualität des Dokuments ausreichend (d. h. ist das Dokument semantisch korrekt) und 3. Ist das Dokument mit dem Vorgängerdokument konsistent. In der Checkliste werden zu jedem Aspekt Fragen gestellt. Werden Fragen in der Checkliste mit Nein beantwortet, dann könnte es sich bei einem Aspekt um einen Fehler handeln.

Abbildung 62 zeigt einen Ausschnitt aus einer *Checkliste* zur Überprüfung der Klassen im E-Klassendiagramm.

Beispiel *Checkliste*

Überprüfe jede Klasse im Klassendiagramm auf:

Einhaltung der Dokumentationsrichtlinien:

1. Wurden die Namenskonventionen eingehalten?
2. *Wenn es sich bei der Klasse um eine aktive Klasse handelt:* Wurde die Klasse von der Klasse ScheduledFct abgeleitet?
3. *Wenn es sich bei der Klasse um eine Sensor- oder Aktuatorklasse handelt:* Wurde die Klasse von der Klasse Portbaustein abgeleitet?
4. *Wenn ein Attribut einer Klasse vom Benutzer gesetzt werden kann:* Kann der Wert vom Control Panel an die Klasse weitergeleitet werden (durch Assoziationen und/oder Durchreichmethoden)?
5. Existiert für jede Klasse mindestens ein Konstruktor?
6. *Wenn durch ein Objekt Speicherplatz allokiert wird (z. B. ein Objekt legt ein anderes an):* Gibt es einen Destruktor für das Objekt, der den Speicherplatz beim Löschen des Objekts wieder freigibt?

Interne Konsistenz der Modelle:

1. *Wenn die Klasse dynamisches Verhalten besitzt:* Existiert ein Zustandsdiagramm für die Klasse?
2. Werden die Assoziationen der Klasse zu anderen Klassen im Zustandsdiagramm verwendet?

Konsistenz zum Klassendiagramm der E-Anforderungen:

1. Existiert zu jeder Klasse, jeder Methode und jedem Attribut aus der Analyse eine Repräsentation im Entwurf?
2. Lässt sich jede Klasse im Entwurf aus den E-Anforderungen oder aus Entwurfsentscheidungen heraus ableiten?

ABBILDUNG 62. Ausschnitt aus der E-Checkliste (UML-Entwurf/Komponenten-Anforderungen)

Fehlerklassifikation

Die aufgedeckten Fehler werden in einer Fehlertabelle aufgelistet. Dabei wird jeder *Fehler* einer *Fehlerklasse* zugeordnet. Die *Fehlerklassifikation* ist identisch zur Fehlerklassifikation bei der Inspektion der E-Anforderungen (siehe Abbildung 32, „Ausschnitt aus einer Fehlerklassifikation - Spezifische Fehlerklassen,“ auf Seite 60).

E-Fehlertabellen

In den *Fehlertabellen* werden alle *Fehler*, die im Produkt *Software-System-Dokumentation* gefunden wurden und noch nicht auf nachfolgende Produkte übertragen wurden, aufgelistet. Das bedeutet, dass die *E-Fehlertabelle* Fehler umfasst, die in der *Software-System-Dokumentation* aufgedeckt worden sind. Die Fehler können ihren Ursprung in der Problemlösungsdokumentation haben. Beispielsweise beinhaltet die *Fehlertabelle* einen *Fehler* im E-Klassendiagramm, der während der Inspektion des UML-Entwurfs gefunden wurde. Ein Entwurfsfehler, der die Ursa-

che für einen Kodefehler ist und nur durch den Kodefehler aufgedeckt wurde, wird nicht in der *E-Fehlertabelle* beschrieben.

Ein in der *E-Fehlertabelle* dokumentierter Fehler kann während der Inspektion des UML-Entwurfs aufgedeckt worden sein. Er kann aber auch in nachfolgenden Prozessen gefunden worden sein, z. B. beim Implementieren. Der Prozess, während dessen ein *Fehler* gefunden wurde, ist in der *Fehlertabelle* vermerkt (z. B. “sysentrevvorb” als Abkürzung für “System-Entwurf Review vorbereiten”). Die *Fehler* werden dabei mit Hilfe des Produkts und des Teilsystems, in dem sie aufgetreten sind, gekennzeichnet (*Produkt*-*Teilsystem*-FNr-*Nummer*). Ein *Fehler* wird durch das betroffene Diagramm und das betroffene Element sowie eine Fehlerbeschreibung charakterisiert.

Abbildung 63 zeigt einen Ausschnitt aus einer *Fehlertabelle* für das Teilsystem “LichtregelungFlur”.

Beispiel *E-Fehlertabellen*

Die *Fehlertabelle* dokumentiert zwei *Fehler* im Teilsystem “LichtregelungFlur” im UML-Entwurf. Beide *Fehler* befinden sich in der Beschreibung der statischen Struktur des Systems (Klassendiagramme). Sie wurden beim Inspizieren des UML-Entwurfs und der Komponenten-Anforderungen gefunden.

FNr	betroffenes Diagramm	betroffenes Element	allgemeine Fehlerbeschreibung	Fehlerklasse	Prozess, während-dessen der Fehler gefunden wurde (Abkürzung)
E-LF-FNr1	Klassendia-gramme	LightController	requestLightStatus: Rückgabetyp nicht festgelegt	U	sysentrevvorb
E-LF-FNr2	Klassendia-gramme	PresenceController-Hallway	Rückgabetypen nicht festgelegt	U	sysentrevvorb

ABBILDUNG 63. Ausschnitt aus der Fehlertabelle Lichtregelung Flur

E-Fehlerbehebungsstrategien

In der *Tabelle zur Dokumentation der Fehlerbehebungsstrategien* wird verdeutlicht, welche Fehler aus der Fehlerliste behoben wurden und wie dies geschehen ist. Außerdem wird in den *Tabellen* die Behebung von Entwurfsfehlern dokumentiert, die die Ursache für Fehler in folgenden Entwicklungsprodukten waren und erst dort entdeckt wurden. Diese Fehler sind in den K-Fehlertabellen dokumentiert. Die *Tabelle* gibt damit Aufschluss über den Stand der Rework-Aktivitäten. Anhand der Tabelle kann außerdem entschieden werden, ob eine erneute Verifikation des geänderten Entwurfs notwendig ist.

Abbildung 64 zeigt einen Ausschnitt aus einer *Tabelle zur Dokumentation der Fehlerbehebungsstrategien* für das Teilsystem “LichtregelungFlur”.

Beispiel *Fehlerbehebungsstrategien*

In der *Tabelle* werden alle Fehler aus der Fehlertabelle aufgeführt. Außerdem dokumentiert die *Tabelle* die Behebung eines Entwurfsfehlers (Ursprung des Fehlers waren bereits die System-Anforderungen), der erst im Kode entdeckt wurde (K-LF-FNr3, K für Kode).

Die Beschreibung der Strategien zur Behebung der Fehler erlaubt es, nachzuvollziehen, was sich im Entwurf geändert hat.

Referenz auf FNr	Strategie zur Behebung des Fehlers	Behebung des Fehlers durchgeführt? (J/N)
E-LF-FNr1	Rückgabewert ergänzt	J
E-LF-FNr2	Rückgabewerte ergänzt	J
K-LF-FNr3	Aufruf beim Zustandsübergang wurde von SwitchOff() in SwitchOn() geändert.	J

ABBILDUNG 64. Ausschnitt aus der Tabelle zur Dokumentation der Fehlerbehebungsstrategien - Teilsystem Lichtregelung Flur

E-Validation

Das Produkt *E-Validation* umfasst alle Dokumente, die für die *Validation* ausführbarer Komponenten (d. h. Klassen) gegen den UML-Entwurf und die Komponenten-Anforderungen wichtig sind. Dies sind: 1. eine Beschreibung des durchgeführten *Testverfahrens*, 2. die aufgestellten *Testpläne*, 3. die *Ergebnisse der durchgeführten Tests* sowie 4. eine *Bewertung der Testergebnisse* und Dokumentation von Fehlverhalten.

Es ist wichtig, Testpläne aufzustellen und durchgeführte Tests und Testergebnisse zu dokumentieren, um gegebenenfalls Tests wiederholen zu können (Regressionstest).

Beim Komponententest wird jede Komponente (d. h. Klasse) im System getestet. Zum Test wird ein Testrahmen benötigt: Ein Treiber aktiviert die zu testende Klasse. Stubs simulieren Klassen, die von der zu testenden Klasse aufgerufen werden.

Dokumentation des Testverfahrens (E-Testverfahren)

Dieses Dokument beschreibt das *Testverfahren*, mit dessen Hilfe Testpläne ausgewählt wurden. Es gibt, wie zum Test des Systems, verschiedene *Black-box Testverfahren* für den Komponententest, z. B. kann ebenfalls ein Äquivalenzklassen- oder Grenzfallbasierter Test durchgeführt werden. Die Wahl eines *Black-box Testverfahrens* erlaubt das Ableiten von Testplänen aus den Komponenten-Anforderungen. Die ausführbare Komponente wird erst zur Durchführung der Tests benötigt. Das

spezielle *Testverfahren* sollte entsprechend des Projekt- und Systemkontexts ausgewählt werden.

Abbildung 65 zeigt einen Ausschnitt aus der *Dokumentation des Testverfahrens*, das in dem Gebäudeautomationsprojekt verwendet wurde.

Beispiel *Dokumentation des Testverfahrens*

Beim betrachteten Gebäudeautomationssystem wurde für jede Komponente ein Black-box-Test durchgeführt. Basis für den Test waren die Zustands- und Aktivitätsdiagramme der Klassen und deren öffentlichen Methoden. Es wurden folgende Testkriterien zugrunde gelegt:

1. Überdeckung der Kanten des Zustands- und des Aktivitätsdiagramms der zu testenden Klasse.
2. Es gibt Klassen die kein Zustandsdiagramm oder Aktivitätsdiagramm besitzen, weil ihre Aufgabe im wesentlichen im Setzen und Abfragen von Werten besteht. Außerdem gibt es Methoden, die nicht im Zustands- oder Aktivitätsdiagramm verwendet werden. Für diese Fälle sind die Testpläne folgendermaßen zu kreieren bzw. zu erweitern:
   - Methoden ohne Rückgabewert werden wenigstens einmal aufgerufen.
   - Ist der Rückgabewert ein Aufzählungstyp:
     Alle definierten Rückgabewerte der einzelnen Methoden werden wenigstens einmal angenommen.
   - Methoden, die dem Setzen von Werten dienen, werden dadurch überprüft, dass ein gesetzter Wert anschließend direkt oder indirekt wieder abgefragt wird. Sollte eine Abfrage des gerade gesetzten Wertes nicht möglich sein, so wird die Methode nicht getestet.
   - Methoden, die dem Abfragen von Werten dienen, werden überprüft, indem abzufragende Werte vorher gezielt gesetzt werden. Wenn der Rückgabewert einer Methode von Methodenaufrufen von anderen Klassen abhängt, sollen die Methoden der aufgerufenen Klasse entsprechende Werte zur Verfügung stellen.

ABBILDUNG 65. Ausschnitt aus der Dokumentation des Testverfahrens

Dokumentation der Testpläne (E-Testpläne)

Die Anzahl der *Testpläne* wird durch das Testkriterium bestimmt. In einem *Testplans* wird beschrieben, wie der Treiber die zu testende Komponente aufrufen muss (d. h. die Eingaben an die Komponente werden festgelegt). Außerdem wird notiert,

welche Methoden von der zu testenden Komponente aufgerufen werden (d. h. das Verhalten der Komponente wird beschrieben). Das Verhalten lässt sich aus den E-Zustandsdiagrammen, E-Aktivitätsdiagrammen und den Klassenbeschreibungen (Class-Tables) ableiten.

Zu jedem *Testplan* gehört eine kurze Beschreibung der benötigten Stubs und Treiber. Dies erlaubt eine Realisierung der Testumgebung ausschließlich basierend auf der Beschreibung der *Testpläne*.

Abbildung 66 zeigt einen Ausschnitt aus einem *Testplan* für die Klasse LightControlUnit des Gebäudeautomationssystems.

Beispiel *Dokumentation der Testpläne*

Ein *Testplan* ist in der Dokumentation des Gebäudeautomationssystems wie folgt aufgebaut:

1. Die Angabe eines Test*schritt*s erlaubt den leichteren Vergleich von Testausgabe und erwartetem Testergebnis.
2. Die Spalte *"vom Treiber aufgerufene Methoden der zu testenden Klasse"* gibt Anweisungen für die Implementierung des Treibers. Es wird notiert, in welcher Reihenfolge Methoden der zu testenden Klasse vom Treiber aufgerufen werden sollen. Außerdem werden gegebenenfalls Ausgaben des Treibers aufgelistet.
3. Das *erwartete Verhalten der zu testenden Klasse* gibt an, wie die zu testende Klasse reagieren soll, z. B. welche Methode einer weiteren Klasse aufgerufen werden soll.
4. Die *gewünschte Reaktion der zu erstellenden Stubs* gibt Anweisungen zur Implementierung der Stubs, z. B. welche Werte an die zu testende Klasse zurückgeliefert werden sollen.
5. Die Spalte *"überdeckte Kante im Zustands-/Aktivitätsdiagramm"* gibt an, welche Kante oder welcher Zustand des Zustands- oder Aktivitätsdiagramms einer Klasse durch den Schritt des *Testplans* überdeckt wird. Dies erleichtert ein Verstehen und Ändern des *Testplans*.

Klasse LightControlUnit
Testplan zu KompTest1: Der Test testet den Zustandsautomaten der Klasse LightControlUnit

Folgende Stubs sind für den Test erforderlich:

- LightController
  void lightControl(zLightStatusType)
  zLightStatusType requestLightStatus().

Schritt	Vom Treiber aufgerufene Methoden der zu testenden Klasse	Erwartetes Verhalten der zu testenden Klasse	Gewünschte Reaktion der zu erstellenden Stubs	Überdeckte Kante im Zustands-/ Aktivitätsdiagramm
1.	LightControlUnit(lightControllerPtr)			"Init" -> "FM-Betrieb"
		-> LightController.lightControl(cLightOff)	LightController.lightControl(cLightOff) <- Ausgabe: Status auf "Licht aus" gesetzt	Entry-Aktion FM-Betrieb
2.	fmLightControl(cLightOn)	-> LightController.lightControl(cLightOn)	LightController.lightControl <- Ausgabe: Status auf "Licht an" gesetzt	Methode fmLightControl(zLightStatusType): void FM-Betrieb
3.	fmLightControl(cLightOff)	-> LightController.lightControl(cLightOff)	LightController.lightControl <- Ausgabe: Status auf "Licht aus" gesetzt	Methode fmLightControl(zLightStatusType): void FM-Betrieb
4.	occupancyMessage(cPersonIn)	-> LightController.lightControl(cLightOn)	LightController.lightControl <- Ausgabe: Status auf "Licht an" gesetzt	FM-Betrieb -> Automatik-Betrieb
5.	changedLightStatus()			Automatik-Betrieb -> manueller Betrieb
6.	occupancyMessage(cPersonOut)	-> LightController.lightControl(cLightOff)	LightController.lightControl <- Ausgabe: Status auf "Licht aus" gesetzt	manueller Betrieb -> FM-Betrieb Entry-Aktion

ABBILDUNG 66. Ausschnitt aus einem Testplan zum Test der Komponente "LightControlUnit"

Dokumentation der Testergebnisse (E-Testergebnisse)

Das Dokument E-Validation erlaubt die *Dokumentation der Testergebnisse* an der gleichen Stelle, an der auch die Testpläne dokumentiert sind. Auf diese Weise können die Testergebnisse mit dem erwartetes Verhalten der Komponente (abgeleitet aus dem Entwurf und dokumentiert im Testplan) leicht miteinander verglichen werden.

Die Testergebnisse beschreiben das Verhalten der Komponente beim Testdurchlauf.

Abbildung 67 zeigt eine mögliche *Dokumentation der Ergebnisse eines durchgeführten Tests* anhand des oben aufgeführten Testplans.

Beispiel *Dokumentation der Testergebnisse*

Ergebnis zu KompTest1 (Korrekter Testlauf)

```
------------------------------------------------
Ausführung von KompTest1 zur Klasse LightControlUnit
------------------------------------------------

Verwendete Stubs:

LightController
 ->lightControl(zLightStatusType):void

------------------------------------------------
------------------ Testlauf beginnt --------------
------------------------------------------------

LightController(1, 0)
Schritt 1: Aufruf des LightControlUnit-Konstruktors
1. Aufruf: Methode LightController->lightControl
>> Status auf “Licht aus” gesetzt

Schritt 2: Aufruf der Methode fmLightControl(cLightOn)
2. Aufruf: Methode LightController->lightControl
>> Status auf “Licht an” gesetzt

Schritt 3: Aufruf der Methode fmLightControl(cLightOff)
3. Aufruf: Methode LightController->lightControl
>> Status auf “Licht aus” gesetzt

Schritt 4: Aufruf der Methode occupancyMessage(cPersonIn)
4. Aufruf: Methode LightController->lightControl
>> Status auf “Licht an” gesetzt

Schritt 5: Aufruf der Methode changedLightStatus

Schritt 6: Aufruf der Methode occupancyMessage(cPersonOut)
5. Aufruf: Methode LightController->lightControl
>> Status auf “Licht aus” gesetzt

------------------------------------------------
------------------ Testlauf beendet --------------
------------------------------------------------
```

ABBILDUNG 67. Ausschnitt aus der Dokumentation zum Test der Komponente “LightControlUnit”

E-Ergebnisbewertung

Nachdem die Tests anhand der Testpläne durchgeführt und die *Testergebnisse* dokumentiert sind, kann die Auswertung der Tests erfolgen. Dafür werden die *Testergebnisse* mit dem im Testplan dokumentierten Ausgabeverhalten (d. h. Ausgaben und Aufrufe von Stubs) der Komponente verglichen. Die Ergebnisse der Auswertung, d. h. entdeckte Fehlverhalten der Komponente, werden mit Hilfe einer *Tabelle* dokumentiert.

Abbildung 68 zeigt einen Ausschnitt aus einer *Tabelle zur Dokumentation von gefundenen Fehlverhalten.*

Beispiel *E-Ergebnisbewertung*

Die Fehlverhalten werden zur leichteren Referenzierung durchnummeriert, wobei die Nummer den Validationsschritt (KV-Komponentenvalidation) und das Teilsystem (LF-LichtregelungFlur) enthält. Die Angabe des Schritts im Testplan erlaubt das leichtere Nachvollziehen der aufgedeckten Fehlverhalten.

FV-Nr	Verweis aus KompTest	Schritt im KompTest	Beschreibung des Fehlverhaltens	Verweis auf Fehler-Nr.
KV-LF-FVNr1	KompTest3	Schritt 6	Das Licht wird ausgeschaltet anstelle angeschaltet.	K-LF-FNr 23
KV-LF-FVNr2	KompTest7	Schritt 4, 5, 7, 8, 9	Der Methodenaufruf “Timer.getCurrentTime()” fehlt (Schritt 4 und 8), wobei in falsche Zustände gesprungen wird (Schritt 5, 7, 9).	K-LF-FNr 25 K-LF-FNr 26

ABBILDUNG 68. Ausschnitt aus der Tabelle zur Bewertung der Testergebnisse

Die beim Test entdeckten Fehlverhalten lassen sich auf Fehler im Kode zurückführen und werden in der K-Fehlertabelle im Dokument K-Verifikation dokumentiert. In die Tabelle zur Bewertung der Testergebnisse wird ein Verweis auf die Fehlernummern ergänzt.

Namenskonventionen

Die *Namenskonventionen* im Prozess “System-Entwurf bearbeiten” erweitern die Namenskonventionen aus dem Prozess “System-Anforderungen bearbeiten”. Sie gewährleisten die Namensgleichheit zu späteren Entwicklungsprodukten und erleichtern eine Abstimmung im Team.

Abbildung 69 zeigt mögliche *Namenskonventionen* für den Prozess “System-Entwurf bearbeiten”.

Beispiel *Namenskonventionen*

- ... (Konventionen aus Prozess “System-Anforderungen bearbeiten”)
- Konstanten beginnen mit einem c.
- Verwendete Typen werden als solche gekennzeichnet. Sie erhalten als Endung Type.
- Zustände beginnen mit einem z.

ABBILDUNG 69. Namenskonventionen für den Prozess “System-Entwurf bearbeiten”

4.3 Rollen

Im Prozess *System-Entwurf bearbeiten* sind die Rollen *Entwurfsingenieur*, *Verifizierer*, *Validierer, Anforderungsingenieur* und *Qualitätsmanager* involviert. Ihre Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 70. Folgende wesentliche Tätigkeiten sind im Prozess *System-Entwurf erstellen* mit den Rollen verbunden:

- ***Entwurfsingenieur.*** Die Aufgabe des Entwurfsingenieurs sind, die *Entwurfsentscheidungen* zu erstellen, in Interaktion mit dem Anforderungsingenieur den *UML-Entwurf* zu erstellen, die *Komponenten-Anforderungen* zu erstellen sowie die *Verfolgbarkeitsmatrix-Entwurf* zu erstellen. Des Weiteren nimmt er an der Verifikation des Entwurfs teil und ist für die Fehleridentifikation und -behebung in den ihm zugeordneten Dokumenten verantwortlich.
- ***Verifizierer.*** Die Aufgaben des Verifizierers sind, die Inspektion des System-Entwurfs zu organisieren, diese in Kooperation mit dem Entwurfsingenieur durchzuführen und die Ergebnisse zu dokumentieren.

- ***Validierer.*** Die Aufgabe des Validierers im Prozess *System-Entwurf bearbeiten* ist, Komponenten-Testfälle zu entwickeln und zu dokumentieren.
- ***Anforderungsingenieur.*** Die Aufgaben des Anforderungsingenieurs sind, den Entwurfsingenieur bei der Erstellung des UML-Entwurfs zu unterstützen sowie bei der Verifikation gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben. Für letzteres kann er den Kunden kontaktieren, sofern die *Problembeschreibung* oder die *Informellen B-Anforderungen* betroffen sind.
- ***Qualitätsmanager.*** Die Aufgabe des Qualitätsmanagers ist, entsprechend den längerfristigen Bedürfnissen und Erwartungen einer Software-Entwicklungsorganisation Wissen/Erfahrung in geeigneter Form zu verwalten, einzelnen Projekten auf Anfrage zur Wiederverwendung zur Verfügung zu stellen und kontinuierlich aufgrund gemachter Projekterfahrungen zu verbessern. Wissen/Erfahrungen kann vom Qualitätsmanager im Prozess *System-Entwurf bearbeiten* z. B. in Form von Checklisten, Namenskonventionen, Fehlerklassifikationen oder wiederverwendbarer Komponenten zur Verfügung gestellt werden. Diese sind zum Teil Bestandteil von Produkten, für die andere Rollen verantwortlich sind. Eigenverantwortlich ist der Qualitätsmanager hier für das Produkt *Namenskonventionen*.

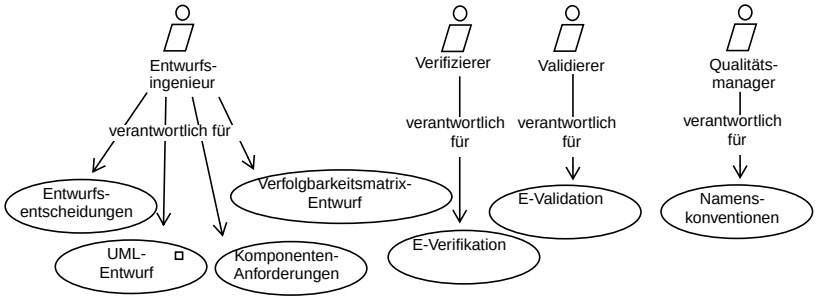


ABBILDUNG 70. Ressourcen und Produkte des Prozesses *System-Entwurf bearbeiten*

4.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 71 dargestellt. Die Beziehung zwischen den Prozessen *UML-Entwurf bearbeiten* und *Komponenten-Anforderungen bearbeiten* ist keine strikt sequenzielle Abfolge. Bei der Bearbeitung der Komponenten-Anforderungen erfolgt eine Vervollständigung des UML-Entwurfs, der daraufhin möglicherweise geändert werden muss. Letzteres geschieht wiederum im Prozess *UML-Entwurf bearbeiten*. Die Folgeprozesse können erst begonnen werden, wenn diese beiden Prozesse vollständig beendet sind.

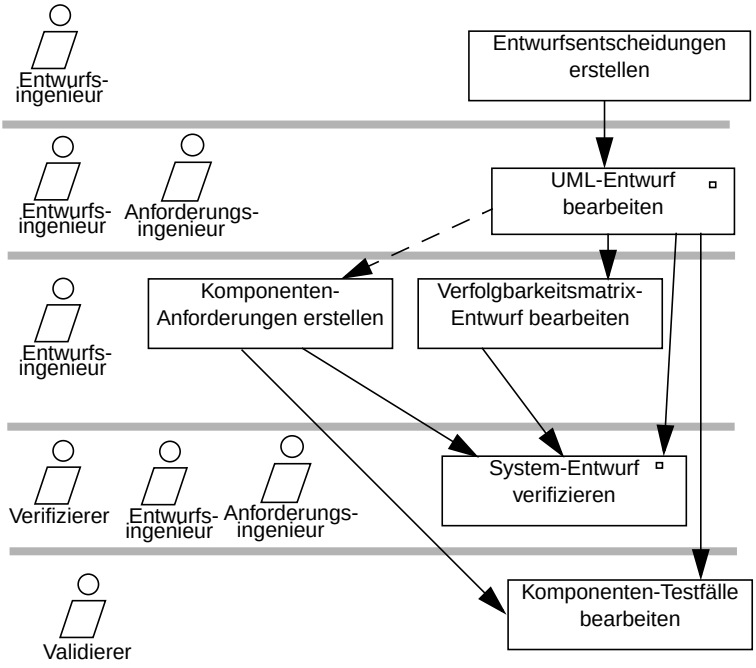


ABBILDUNG 71. Kontrollfluss für den Prozess *System-Entwurf bearbeiten*

4.5 Prozess “Entwurfsentscheidungen erstellen”

Im Prozess *Entwurfsentscheidungen erstellen* werden aus den informellen nicht-funktionalen B-Anforderungen und dem Wissen über die Systemumgebung Entwurfsentscheidungen abgeleitet.

Die informellen nicht-funktionalen B-Anforderungen geben Richtlinien für die Strukturierung des Systems vor. Beispielweise führt die nicht-funktionale Anforderung der Änderbarkeit eines Systems zu einer bestimmten Systemstruktur. Dafür muss im Vorfeld geklärt werden, welche Bereiche des Systems (z. B. welche Anforderungen, welche Schnittstellen zu anderen Systemen oder welche Strukturelemente aus der Umgebung) stabil sind und welche häufigen Änderungen unterworfen sind. Bereiche, deren Änderung abzusehen ist, sollten in Komponenten (d. h. Teilsystemen oder Klassen) gekapselt (und nicht auf das gesamte System verteilt) werden. So minimiert man die Auswirkung von Änderungen auf das System.

Neben den informellen nicht-funktionalen B-Anforderungen liefert die Systemumgebung Einschränkungen für die Implementierung. Zum Beispiel macht der verwendete Zielrechner Einschränkungen hinsichtlich der Parallelität (z. B. wenn das Betriebssystem keine Parallelität erlaubt) und Effizienz (z. B. das System muss mit einem bestimmten Speicher auskommen).

Eventuell stellt der Qualitätsmanager Komponenten bereit, die wiederverwendet werden können. Durch die Wiederverwendung von Komponenten wird die Implementierung ebenfalls eingeschränkt.

Die gesammelten Entwurfsentscheidungen werden dokumentiert.

Abbildung 72 zeigt die verwendeten Produkte im Prozess *Entwurfsentscheidungen erstellen*.

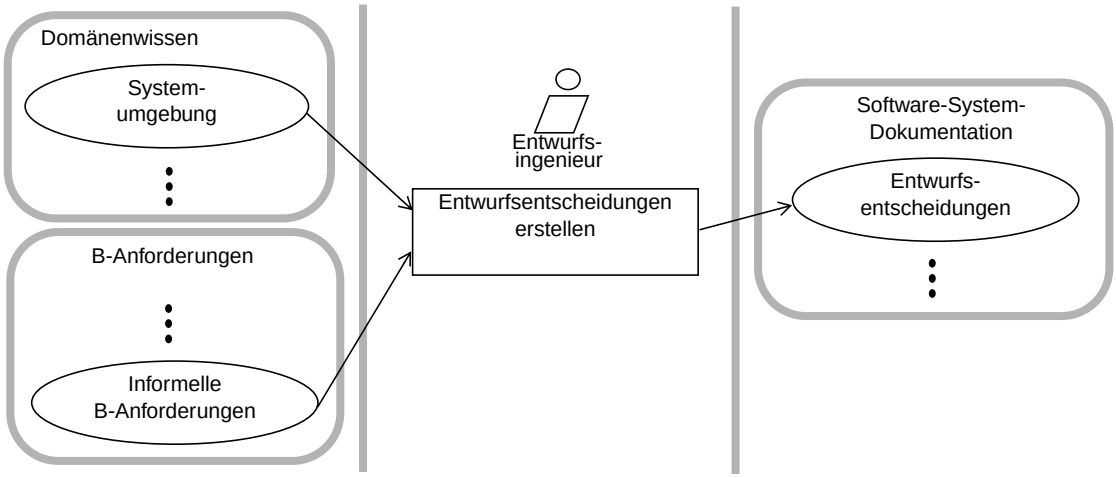


ABBILDUNG 72. Verwendete Produkte des Prozesses *Entwurfsentscheidungen erstellen*

Folgende Produkte werden im Prozess *Entwurfsentscheidungen erstellen* konsumiert:

- Systemumgebung.
- Informelle B-Anforderungen.

Folgendes Produkt wird produziert:

- Entwurfsentscheidungen.

4.6 Prozess "UML-Entwurf bearbeiten"

Im Prozess *UML-Entwurf bearbeiten* werden die Diagramme aus den E-Anforderungen hinsichtlich der Entwurfsentscheidungen und anderer Entwurfsaspekte überarbeitet und verfeinert. Die Realisierbarkeit des Software-Systemmodells wird gewährleistet und die endgültige Struktur des Software-Systems festgelegt. Insbesondere bei der Verfeinerung helfen die Namenskonventionen, Bezeichner (z. B. von Konstanten oder Datentypen) zu benennen.

Der Prozess *UML-Entwurf bearbeiten* geschieht in Interaktion mit dem Anforderungsingenieur, der die Anforderungen erarbeitet hat. Hauptziel ist es, eine Lösungsstruktur für die Anforderungen festzulegen.

Abbildung 73 zeigt die verwendeten Produkte im Prozess *UML-Entwurf bearbeiten*.

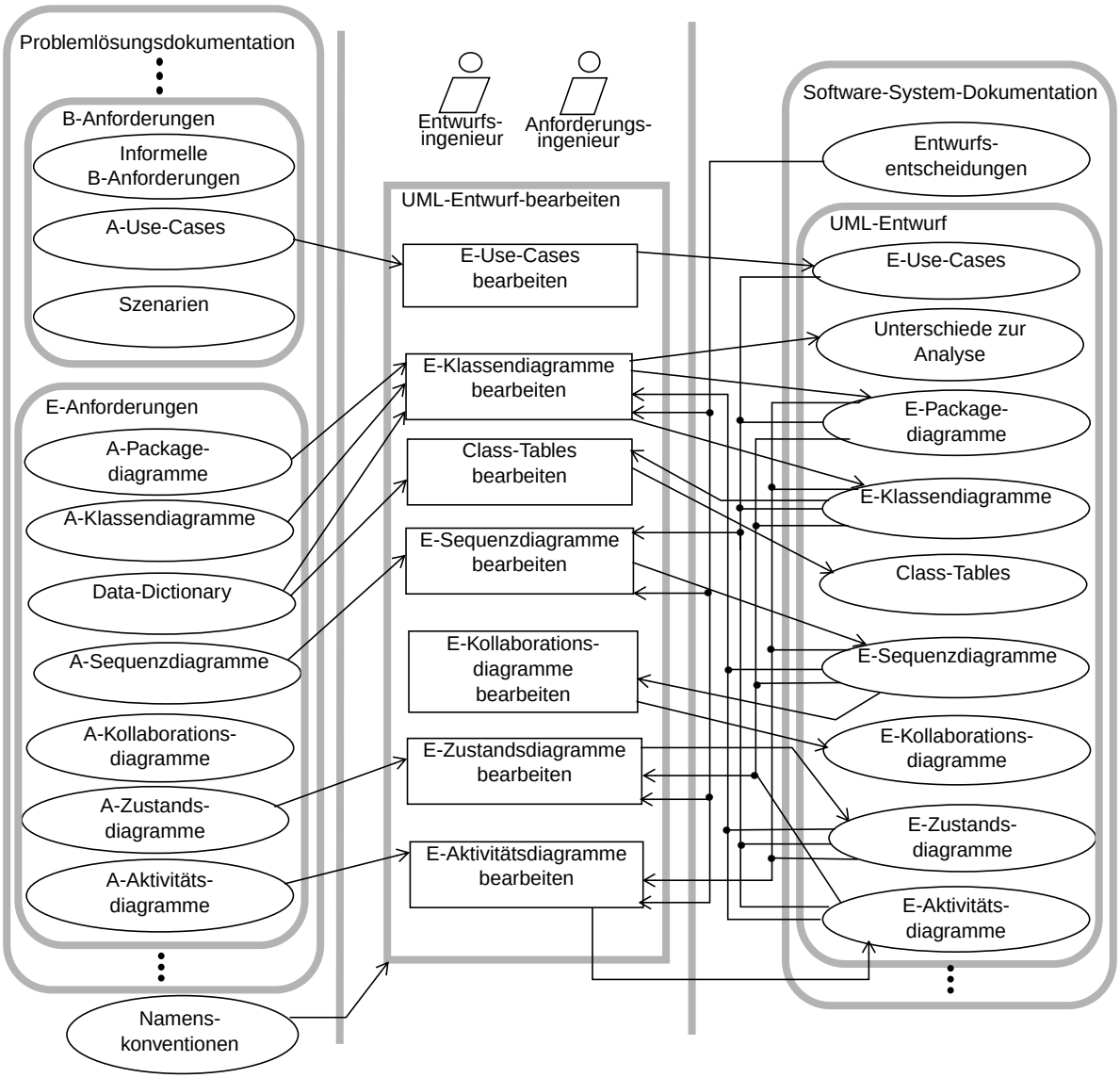


ABBILDUNG 73. Verwendete Produkte des Prozesses *UML-Entwurf bearbeiten*

Folgende Produkte werden im Prozess *UML-Entwurf bearbeiten* konsumiert:

- A-Use Cases. Das Dokument beschreibt die möglichen Verwendungsweisen des zu entwickelnden Systems.
- A-Packagediagramme. Das Dokument beschreibt eine Zerlegung des Systems in Teilsysteme.
- A-Klassendiagramme. Das Dokument beschreibt Klassen des Systems mit ihren Attributen, Methoden und Beziehungen.
- Data-Dictionary. Das Dokument erläutert die verschiedenen Elemente des A-Klassendiagramms informell.
- A-Sequenzdiagramme. Das Dokument beschreibt das Verhalten des Systems als Interaktion von Objekten der Klassen aus dem Klassendiagramm.
- A-Zustandsdiagramme. Das Dokument beschreibt das Verhalten einzelner Klassen.
- A-Aktivitätsdiagramme. Das Dokument verfeinert einzelne komplexe Operationen.
- Entwurfsentscheidungen. Das Dokument beschreibt Implementierungsentscheidungen, die aus der Systemumgebung oder Qualitätsanforderungen resultieren.
- Namenskonventionen. Das Dokument beschreibt Richtlinien für die Wahl von Bezeichnern.

Folgendes Produkt wird produziert:

- UML-Entwurf. Das Dokument beinhaltet die überarbeiteten und verfeinerten UML-Diagramme. Die E-Klassen- und Packagediagramme beschreiben die Struktur des zu implementierenden Systems. Unterschiede zwischen dem A- und dem E-Klassendiagramm werden im Dokument “Unterschiede zur Analyse” dokumentiert. Die Klassenbeschreibungen (Class-Tables) beschreiben die Klassen, Attribute, Methoden und Beziehungen aus den Klassendiagrammen. Die E-Sequenz- und E-Kollaborationsdiagramme beschreiben das Verhalten des Systems durch die Interaktion von Objekten. Die E-Zustandsdiagramme beschreiben das Verhalten einzelner Klassen. Die E-Aktivitätsdiagramme verfeinern einzelne Operationen.

Die Teilprozesse E-Klassendiagramme bearbeiten, E-Sequenzdiagramme bearbeiten, E-Zustandsdiagramme und E-Aktivitätsdiagramme bearbeiten beeinflussen sich gegenseitig und lassen sich nicht unabhängig voneinander durchführen: Werden z. B. zu einer Methode im Klassendiagramm Parameter und Rückgabewerte

angegeben, dann muss diese Verfeinerung auch im Sequenz- und Zustandsdiagramm nachvollzogen werden. Die Konsistenz zwischen den verschiedenen Diagrammen muss gewährleistet werden. Zu jedem dieser Teilprozesse werden im weiteren Beispiele für Konsistenzbedingungen angegeben. Zum Teil werden Konsistenzchecks zwischen den Diagrammen von existierenden CASE-Werkzeugen übernommen.

Prozess *"E-Use Cases bearbeiten"*

Im Teilprozess *E-Use Cases bearbeiten* werden die A-Use Cases aus den B-Anforderungen in den UML-Entwurf unverändert übernommen. Sie dienen im UML-Entwurf zur Einordnung der E-Sequenzdiagramme.

Folgendes Produkt wird im Prozess *E-Use Cases bearbeiten* konsumiert:

- A-Use Cases.

Folgendes Produkt wird produziert:

- E-Use Cases.

Prozess *"E-Klassendiagramme bearbeiten"*

Während des Prozesses *E-Klassendiagramme bearbeiten* werden die A-Packagediagramme und die A-Klassendiagramme aus den E-Anforderungen verfeinert und hinsichtlich der Entwurfsentscheidungen und anderer Entwurfskriterien überarbeitet. Änderungen in den Klassendiagrammen (durch Anpassungen an Entwurfsentscheidungen oder Anwendung von Optimierungskriterien) werden im Dokument "Unterschiede zur Analyse" der Software-Systemdokumentation erläutert. Dies erlaubt das leichtere Nachvollziehen von getroffenen Entwurfsentscheidungen z. B. im Fall von späteren Änderungen.

Die Namenskonventionen geben Richtlinien vor für die Wahl von Bezeichnern bei Verfeinerungen (z. B. Konstanten beginnen mit c).

Die Konsistenz der E-Klassendiagramme zu den anderen UML-Diagrammen muss sichergestellt sein. Dafür muss überprüft werden, ob Änderungen in den anderen Diagrammen in den Klassendiagrammen nachgezogen wurden (z. B. wenn zwei Methoden zu einer Methode mit Parametern im Sequenzdiagramm zusammengefasst wurden, muss diese Änderung auch im Klassendiagramm nachvollzogen werden). Die Konsistenzbedingungen aus dem Prozess "A-Klassendiagramme

bearbeiten” gelten weiterhin und werden evtl. verfeinert, z. B. muss überprüft werden, ob:

- die Methoden mit Parametern und Rückgabewerten, die eine Klasse im Zustandsdiagramm geschickt bekommt (Events), auch als Methoden mit Parametern und Rückgabewerten im Klassendiagramm definiert sind.
- die Methoden mit Parametern und Rückgabewerten, die die Objekte einer Klasse in den Sequenzdiagrammen geschickt bekommen, auch als Methoden mit Parametern und Rückgabewerten im Klassendiagramm definiert sind.

Folgende Produkte werden im Prozess *E-Klassendiagramme bearbeiten* konsumiert:

- A-Klassendiagramme.
- A-Packagediagramme.
- Data-Dictionary.
- Entwurfsentscheidungen.
- E-Sequenzdiagramme.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.

Folgende Produkte werden produziert:

- E-Packagediagramme.
- E-Klassendiagramme.
- Unterschiede zur Analyse.

Prozess *“Klassenbeschreibungen (Class-Tables) bearbeiten”*

Im Teilprozess *Class-Tables-bearbeiten* werden die Klassen aus dem E-Klassendiagramm mit ihren Attributen und öffentlichen Methoden erklärt. Einige CASE-Werkzeuge erlauben das Dokumentieren einzelner Diagrammelemente. Wird diese Funktionalität genutzt, können die Class-Tables mit Hilfe des CASE-Werkzeugs (z. B. StP/UML) generiert werden.

Folgende Produkte werden im Prozess *Klassenbeschreibungen (Class-Tables) bearbeiten* konsumiert:

- E-Klassendiagramme.

- Data-Dictionary.

Folgendes Produkt wird produziert:

- Class-Tables.

Prozess *"E-Sequenzdiagramme bearbeiten"*

Im Prozess *E-Sequenzdiagramme bearbeiten* wird das A-Sequenzdiagramm aus den E-Anforderungen verfeinert und hinsichtlich der Entwurfsentscheidungen und anderer Entwurfskriterien überarbeitet. Die Namenskonventionen geben Richtlinien für die Wahl von Bezeichnern.

Die Konsistenz der E-Sequenzdiagramme zu den anderen UML-Diagrammen muss sichergestellt sein. Dafür muss überprüft werden, ob Änderungen in den anderen UML-Diagrammen in den Sequenzdiagrammen nachgezogen wurden. Die Konsistenzbedingungen aus dem Prozess "A-Sequenzdiagramme bearbeiten" gelten weiterhin und werden evtl. verfeinert, z. B. muss überprüft werden, ob:

- die Methoden mit Parametern und Rückgabewerten einer Klasse im Klassendiagramm in den Sequenzdiagrammen an Objekte der Klasse mit Parametern und Rückgabewerten geschickt werden.
- die Methoden mit Parametern und Rückgabewerten, die im Zustandsdiagramm als Aktionen modelliert sind, von Objekten der Klasse in den Sequenzdiagrammen verschickt werden.

Folgende Produkte werden im Prozess *E-Sequenzdiagramme bearbeiten* konsumiert:

- A-Sequenzdiagramme.
- Entwurfsentscheidungen.
- E-Use-Cases.
- E-Packagediagramme.
- E-Klassendiagramme.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.
- Namenskonventionen.

Folgendes Produkt wird produziert:

- E-Sequenzdiagramme.

Prozess *“E-Kollaborationsdiagramme bearbeiten”*

Im Prozess *E-Kollaborationsdiagramme bearbeiten* werden die E-Kollaborationsdiagramme aus den E-Sequenzdiagrammen abgeleitet.

Mit CASE-Werkzeugen (z. B. StP/UML) können aus den Sequenzdiagrammen Kollaborationsdiagramme automatisch generiert werden. Die Generierung sollte geschehen, wenn sich die Sequenzdiagramme in einem stabilen Zustand befinden.

Folgendes Produkt wird in dem Prozess *E-Kollaborationsdiagramme bearbeiten* konsumiert:

- E-Sequenzdiagramme.

Folgendes Produkt wird produziert:

- E-Kollaborationsdiagramme.

Prozess *“E-Zustandsdiagramme bearbeiten”*

Im Prozess *E-Zustandsdiagramme bearbeiten* werden die A-Zustandsdiagramme aus den E-Anforderungen verfeinert und hinsichtlich der Entwurfsentscheidungen und weiterer Entwurfskriterien überarbeitet. Die Namenskonventionen geben Richtlinien für die Wahl von Bezeichnern.

Die Konsistenz der E-Zustandsdiagramme zu den anderen UML-Diagrammen muss sichergestellt sein. Dafür muss überprüft werden, ob Änderungen in den anderen UML-Diagrammen im Zustandsdiagramm nachgezogen wurden. Die Konsistenzbedingungen aus dem Prozess “A-Zustandsdiagramme bearbeiten” gelten weiterhin und werden evtl. verfeinert, z. B. muss überprüft werden, ob:

- die Methoden mit Parametern und Rückgabewerten, die die Objekte einer Klasse in den Sequenzdiagrammen geschickt bekommen, in dem Zustandsdiagramm der Klasse als Events mit Parametern und Rückgabewerten modelliert sind.
- die Methoden mit Parametern und Rückgabewerten, die die Objekte einer Klasse in den Sequenzdiagrammen verschicken, als Aktionen im Zustandsdiagramm modelliert sind.

Folgende Produkte werden im Prozess *E-Zustandsdiagramme bearbeiten* konsumiert:

- A-Zustandsdiagramme.
- Entwurfsentscheidungen.
- E-Packagediagramme.
- E-Klassendiagramme.
- E-Sequenzdiagramme.
- E-Aktivitätsdiagramme.

Folgendes Produkt wird produziert:

- E-Zustandsdiagramme.

Prozess *"E-Aktivitätsdiagramme bearbeiten"*

Im Prozess *E-Aktivitätsdiagramme bearbeiten* werden die A-Aktivitätsdiagramme aus den E-Anforderungen verfeinert und hinsichtlich der Entwurfsentscheidungen und weiterer Entwurfskriterien überarbeitet. Informelle Beschreibungen von Operationen in den E-Anforderungen werden präzisiert. Für komplexe Operationen, die während des System-Entwurfs lokalisiert wurden, werden neue Aktivitätsdiagramme eingefügt.

Die Konsistenz der E-Aktivitätsdiagramme zu den anderen UML-Diagrammen muss sichergestellt sein. Dafür muss überprüft werden, ob Änderungen in den anderen UML-Diagrammen in den Aktivitätsdiagrammen nachgezogen wurden. Die Konsistenzbedingungen aus dem Prozess "A-Aktivitätsdiagramme bearbeiten" gelten weiterhin und werden evtl. verfeinert, z. B. muss überprüft werden, ob:

- die Methoden mit Parametern und Rückgabewerten, die die Objekte einer Klasse in den Sequenzdiagrammen geschickt bekommen, in dem Aktivitätsdiagramm der Klasse als Events mit Parametern und Rückgabewerten modelliert sind.
- die Methoden mit Parametern und Rückgabewerten, die die Objekte einer Klasse in den Sequenzdiagrammen verschicken, als Aktionen im Aktivitätsdiagramm modelliert sind.

Folgende Produkte werden im Prozess *E-Aktivitätsdiagramme bearbeiten* konsumiert:

- A-Aktivitätsdiagramme.

- Entwurfsentscheidungen.
- E-Packagediagramme.
- E-Klassendiagramme.
- E-Sequenzdiagramme.
- E-Zustandsdiagramme.

Folgendes Produkt wird produziert:

- E-Aktivitätsdiagramme.

4.7 Prozess “Komponenten-Anforderungen erstellen”

Im Prozess *Komponenten-Anforderungen erstellen*, wird die Information des UML-Entwurfs vervollständigt. Nach der Fertigstellung der Komponenten-Anforderungen soll eine unabhängige Implementierung der Komponenten möglich sein. Dafür werden die Richtungen von Assoziationen festgelegt und Pointernamen für die Realisierung von Assoziationen und Aggregationen festgelegt. Außerdem werden importierte Datentypen und Methoden pro Klasse aufgelistet. Private Methoden sowie Konstruktoren und Destruktoren werden erläutert.

Die Information über die importierten Methoden kann aus den Sequenzdiagrammen abgeleitet werden.

Die Vervollständigung des UML-Entwurfs kann zu einer Änderung des UML-Entwurfs führen. Beispielsweise wird beim Festlegen der Richtungen für Assoziationen das E-Klassendiagramm verändert.

Abbildung 74 zeigt die im Prozess *Komponenten-Anforderungen erstellen* verwendeten Produkte.

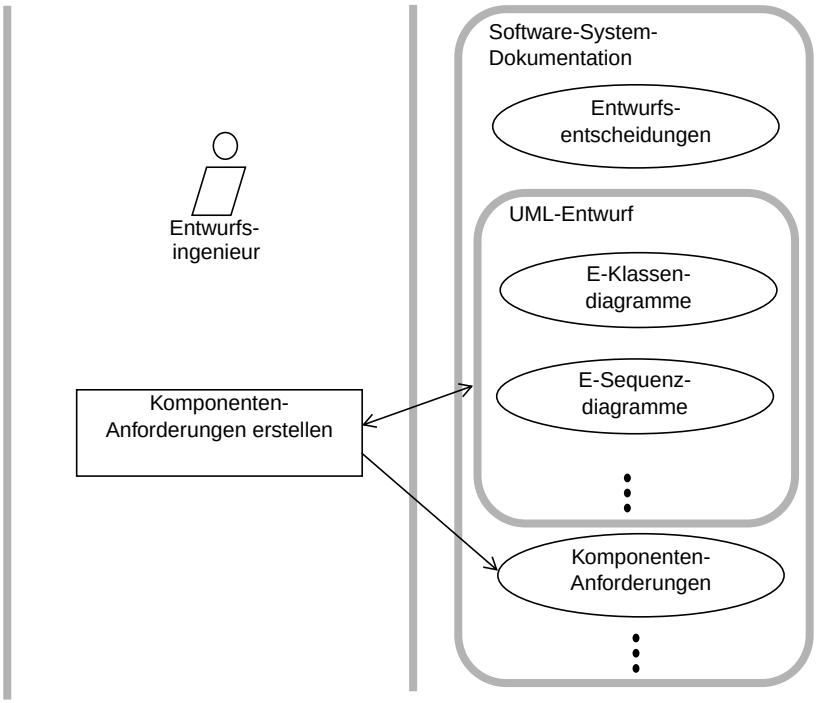


ABBILDUNG 74. Verwendete Produkte des Prozesses *Komponenten-Anforderungen erstellen*

Im Prozess *Komponenten-Anforderungen erstellen* werden folgende Produkte modifiziert:

- UML-Entwurf.

Folgende Produkte werden produziert:

- Komponenten-Anforderungen.

4.8 Prozess “Verfolgbarkeitsmatrix-Entwurf bearbeiten”

Im Prozess *Verfolgbarkeitsmatrix-Entwurf bearbeiten* wird eine Beziehung zwischen den Methoden und Attributen der Klassen des Entwurfs und den Methoden und Attributen der Klassen der E-Anforderungen hergestellt. Für jede Methode/Attribut einer Klasse im Entwurf, die/das an der Realisierung einer Methode/eines Attributs einer Klasse aus den E-Anforderungen beteiligt ist, wird ein Kreuz in die Matrix eingefügt.

Abbildung 75 zeigt den Prozess *Verfolgbarkeitsmatrix-Entwurf bearbeiten* mit konsumierenden und produzierenden/modifizierenden Produkten.

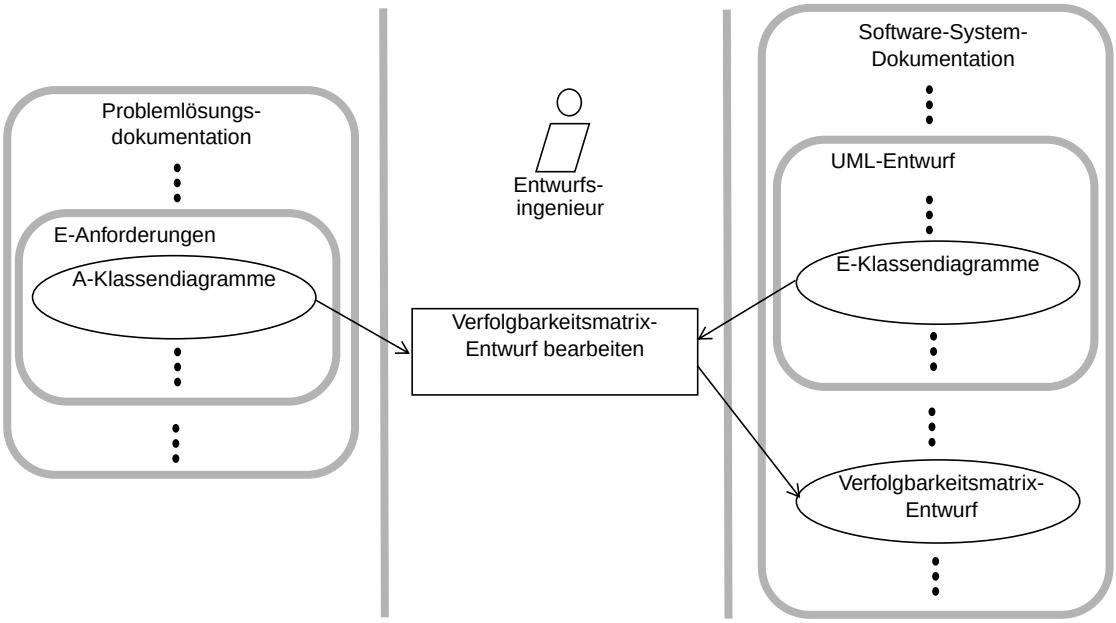


ABBILDUNG 75. Verwendete Produkte des Prozesses *Verfolgbarkeitsmatrix-Entwurf bearbeiten*

Im Prozess *Verfolgbarkeitsmatrix-Entwurf bearbeiten* werden folgende Produkte konsumiert:

- A-Klassendiagramme
- E-Klassendiagramme

Folgendes Produkt wird produziert:

- Verfolgbarkeitsmatrix-Entwurf.

Eine Generierung der Verfolgbarkeitsmatrix-Entwurf aus vorhandenen Verfolgbarkeitsbeziehungen ist grundsätzlich möglich. Im Gebäudeautomationsprojekt konnte die Matrix nicht generiert werden, da die E-Anforderungen und der UML-Entwurf als zwei StP-Systeme verwaltet werden. Beziehungen zwischen zwei Systemen lassen sich unter StP/UML nicht herstellen.

4.9 Prozess "System-Entwurf verifizieren"

Im Prozess *System-Entwurf verifizieren* wird der erstellte Systementwurf, die Komponenten-Anforderungen und die Verfolgbarkeitsmatrix-Entwurf gegen die E-Anforderungen verifiziert. Anleitungen für und Ergebnisse der Verifikation werden im Dokument E-Verifikation dokumentiert.

Im folgenden wird sich auf die Inspektion als Verifikationsaktivität eingeschränkt.

Ein Inspektionsprozess umfasst drei Teilprozesse: 1. Lesen des zu inspizierenden Dokuments (*System-Entwurf Review vorbereiten*), 2. Durchführen einer Inspektionssitzung zur Besprechung aufgedeckter Fehler (*Reviewsitzung System-Entwurf*) und 3. Überarbeiten des verifizierten Dokuments zur Behebung von Fehlern (*Rework System-Entwurf*).

Abbildung 76 zeigt den Prozess *System-Entwurf verifizieren* mit konsumierenden, produzierenden und modifizierenden Produkten.

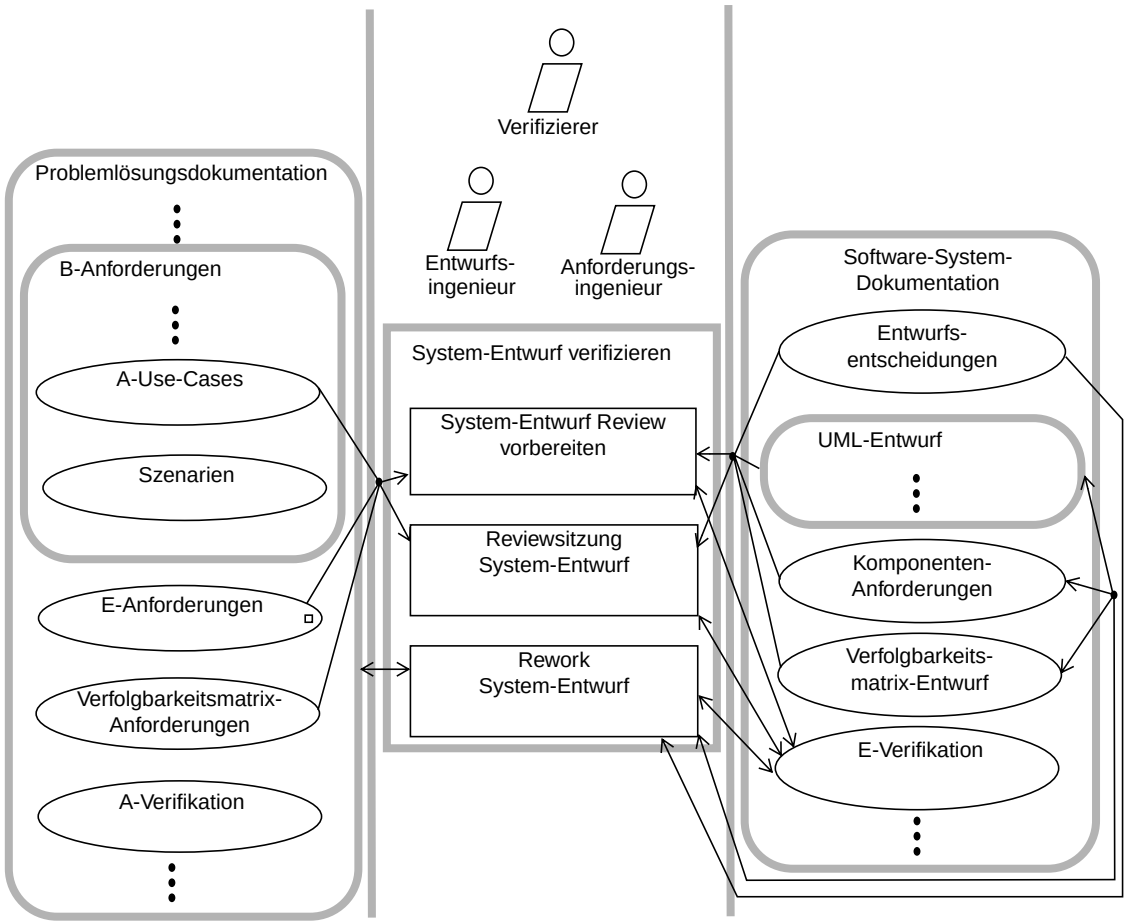


ABBILDUNG 76. Verwendete Produkte des Prozesses *System-Entwurf verifizieren*

Folgende Produkte werden im Prozess *System-Entwurf verifizieren* konsumiert:

- Entwurfsentscheidungen.

Folgende Produkte werden modifiziert:

- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- A Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

Prozess *"System-Entwurf Review vorbereiten"*

Im Prozess *System-Entwurf Review vorbereiten* werden der UML-Entwurf, die Komponenten-Anforderungen und die Verfolgbarkeitsmatrix-Entwurf mit Hilfe der E-Checkliste, die im Dokument E-Verifikation dokumentiert ist, überprüft. Gelesen wird von den Verifizierern. Werden Fragen in der Checkliste mit "Nein" beantwortet, dann könnte es sich bei einem Aspekt um einen Fehler handeln. Probleme in den Dokumenten werden in der Fehlertabelle dokumentiert.

Ergebnis dieses Prozesses ist eine Liste mit möglichen Fehlern. Jeder Fehler ist dabei einer Fehlerklasse zuzuordnen. Die potentielle Fehlerliste wird im Dokument E-Verifikation dokumentiert.

Im Prozess *System-Entwurf Review vorbereiten* werden folgende Produkte konsumiert:

- A-Use Cases.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- Entwurfsentscheidungen.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.

Im Prozess wird folgendes Produkt modifiziert:

- E-Verifikation.

Prozess *“Reviewsitzung System-Entwurf”*

Im Prozess *Reviewsitzung-System-Entwurf* werden die beim Lesen gefundenen Fehler besprochen. An der Inspektionssitzung nehmen Entwurfsingenieure und Verifizierer teil. Die Aufgabe der Verifizierer ist es nicht, die erkannten Fehler zu lösen, sondern lediglich, diese während der Sitzung in den Dokumenten aufzuzeigen. Die Verifizierer führen durch die Liste der Fehler. Ergebnis der Inspektionssitzung ist eine überarbeitete Fehlerliste, die im Dokument E-Verifikation festgehalten wird.

Im Prozess *Reviewsitzung System-Entwurf* werden folgende Produkte konsumiert:

- A-Use Cases.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- Entwurfsentscheidungen.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.

Folgendes Produkt wird modifiziert:

- E-Verifikation.

Prozess *“Rework System-Entwurf”*

Im Prozess *Rework System-Entwurf* werden die Fehler aus der Fehlerliste behoben. Dabei wird die Strategie zur Behebung des Fehlers dokumentiert, um Änderungen an den Dokumenten zu verdeutlichen. Dies geschieht mit Hilfe der Tabelle zur Dokumentation der Fehlerbehebungsstrategie im Dokument E-Verifikation. Lässt sich ein Fehler auf einen Fehler in den System-Anforderungen zurückführen, wird der Fehler auch in den System-Anforderungen behoben. Die Behebung wird im Dokument A-Verifikation beschrieben.

Im Prozess *Rework System-Entwurf* werden folgende Produkte konsumiert:

- Entwurfsentscheidungen.

Folgende Produkte werden modifiziert:

- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.

4.10 Prozess "Komponenten-Testfälle bearbeiten"

Im Prozess *Komponenten-Testfälle bearbeiten* werden entsprechend des ausgewählten Black-box Testverfahrens Testpläne entwickelt. Ein mögliches Testkriterum besteht in der Überdeckung der Kanten der Zustands- und Aktivitätsdiagramme der Klassen sowie ihrer öffentlichen Methoden.

Der Test der ausführbaren Komponente soll basierend auf den Testplänen möglich sein. Für den Test einer Komponente wird ein Testrahmen (Stubs und Treiber) benötigt. Der Testrahmen muss als Teil des Testplans beschrieben sein. Er beschreibt, welche Stubs mit welchen Methoden für den Test erforderlich sind, welche Objekte im Treiber angelegt werden müssen, welche Methoden durch den Treiber aufgerufen werden müssen und welche Rückgabewerte von Stubs bereitgestellt werden sollen. Mit Hilfe des Treibers und der Stubs werden die Eingabewerte für die zu testende Komponente festgelegt.

Außerdem muss im Testplan das erwartete Verhalten einer Komponente beschrieben werden. Beispielsweise wird dokumentiert, welche Methoden die zu testende Komponente aufruft. Die Dokumentation der Testpläne (Ein- und Ausgaben der Komponente) erfolgt mit Hilfe einer Tabelle.

Die Pläne zum Test einer Klasse sollten nicht durch den Entwerfer der Klasse erstellt werden. Auf diese Weise wird gewährleistet, dass der Testplan von einer unabhängigen Partei entwickelt wird.

Abbildung 77 zeigt den Prozess *Komponenten-Testfälle bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

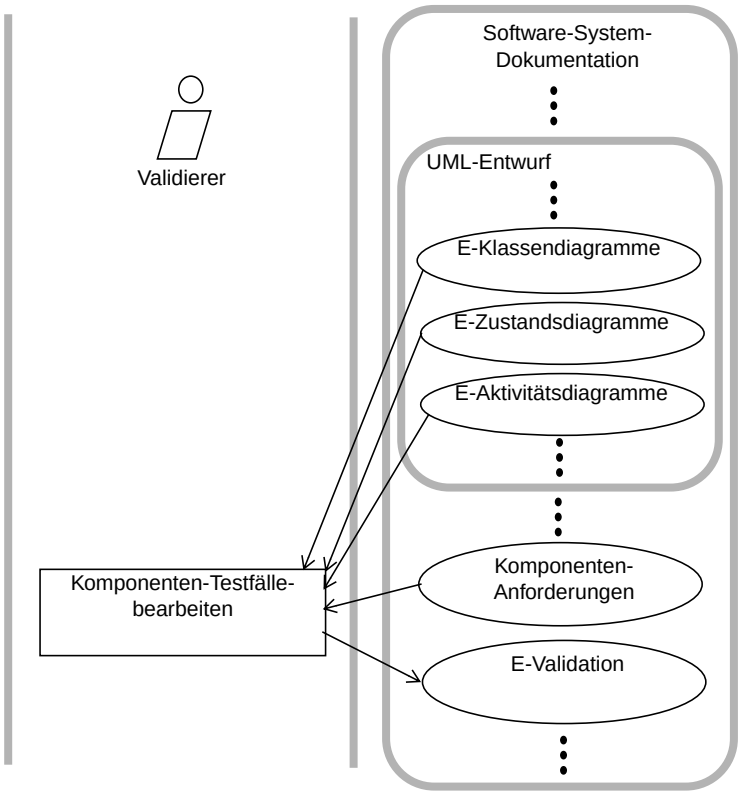


ABBILDUNG 77. Verwendete Produkte des Prozesses *Komponenten-Testfälle bearbeiten*

Folgende Produkte werden im Prozess *Komponenten-Testfälle bearbeiten* konsumiert:

- E-Klassendiagramme.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.
- Komponenten-Anforderungen.

Folgendes Produkt wird produziert:

- E-Validation.

4.11 Weiterführende Literatur

1. A. F. Ackerman, L. S. Buchwald, F. H. Lewski: *Software Inspections: An Effective Verification Process.* In: Donald J. Reifer: *Software Management.* 5th Edition, IEEE Computer Society, 1997, Kapitel 10, S. 416--421.
2. Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice (SEI Series in Software Engineering)*. Addison Wesley Publishing Company, 1997.
3. Grady Booch, James Rumbaugh, Ivar Jacobson: *Das UML-Benutzerhandbuch.* Addison-Wesley Longman, 1999.
4. Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide.* Addison-Wesley Longman, 1999.
5. Pankaj Jalote: *Integrated approach to software engineering*. 2nd ed. Springer Verlag, 1997.
6. Barbara Kitchenham, Steve Linkman: *Validation, Verification, and Testing: Diversity Rules.* IEEE Software, Vol. 15, Nr. 4, Juli 1998, S. 46-49.
7. P. Liggesmeyer, M. Rothfelder, M. Rettelbach, T. Ackermann: *Qualitätssicherung Software-basierter technischer Systeme - Problembereiche und Lösungsansätze.* GI Informatik Spektrum, Vol. 21, Nr. 5, Oktober 1998, S. 249-258.
8. Bertrand Meyer: *Object-oriented Software Construction.* Prentice Hall 1988.
9. Bernd Oesterreich: *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenburg-Verlag, 4., aktualisierte Auflage 1998.
10. James Rumbaugh, Michael Blaha, William Premerali: *Objektorientiertes Modellieren und Entwerfen.* Hanser Elektronik, München, 1993.
11. M. Shaw, D. Garlan: *Software architecture: perspectives on an emerging discipline.* Prentice Hall 1996.

12. Wolfgang Wagenbichler: *Erstellung einer SW-Systemdokumentation mit StP.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 67653 Kaiserslautern, 1999. URL: http://wwwagse.informatik.uni-kl.de/access-system/publications.html#8

KAPITEL 5

System-Komponenten bearbeiten

Ziel des Prozesses *System-Komponenten bearbeiten* ist die Implementierung und Validierung der im UML-Entwurf und den Komponenten-Anforderungen beschriebenen Komponenten. Die Implementierung und Validierung kann für jede Komponente unabhängig erfolgen. Der *Komponenten-Kode* als Ergebnis der Implementierung wird in der *Software-Komponenten-Dokumentation* fixiert.

Abbildung 78 gibt einen Überblick über die verwendeten Produkte im Prozess *System-Komponenten bearbeiten*.

Folgendes Produkt wird konsumiert:

- *Entwicklungsrichtlinien*. Dieses Dokument beinhaltet Richtlinien für die Programmierung. Es umfasst einerseits Vorschriften für die Wahl von Bezeichnern, für das Aussehen des Quellkodes und für dessen Kommentierung. Andererseits umfasst es Vorschriften, wie der *Komponenten-Kode* aus dem UML-Entwurf und den Komponenten-Anforderungen abzuleiten ist.

Folgende Produkte werden im Prozess *System-Komponenten bearbeiten* modifiziert:

- *Software-Komponenten-Dokumentation*. Dieses Dokument umfasst den *Quellkode* der implementierten Komponenten. Darüber hinaus beinhaltet die Dokumentation *Dokumente zu Verifikationsaktivitäten* auf Implementierungsebene (z. B. eine Klassifikation für Fehler, eine Checkliste zur Inspektion des Kodes und Fehlertabellen zur Dokumentation gefundener Fehler und Fehlerursachen).
- *Ausführbare Komponente.* Das Produkt *ausführbare Komponente* enthält den Quellkode der Testrahmen (*Komponententest-Rahmen*), die zum Test der Komponenten benötigt werden. Darüber hinaus umfasst die *ausführbare Komponente* den *ausführbaren Komponenten-Kode*. Dieser enthält den übersetzten Kode der Komponenten und der Testrahmen sowie eine Menge ausführbarer Binaries (d. h. übersetzte und mit einem Testrahmen zusammengebundene Komponenten).
- *Problemlösungsdokumentation.* Dieses Dokument beinhaltet die Anforderungen an das zu entwickelnde Software-System aus Sicht der Benutzer/Kunden und der Entwickler.
- *Software-System-Dokumentation.* Dieses Dokument umfasst den UML-Entwurf für das zu implementierende Software-System. Außerdem beinhaltet es Anforderungen an die zu implementierenden Komponenten im System.

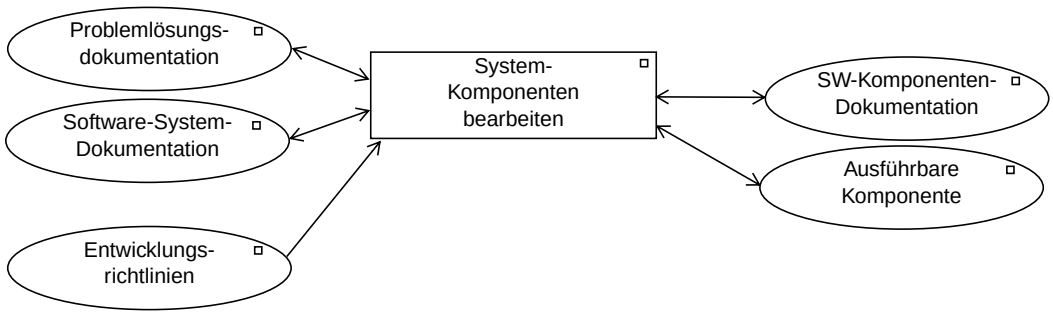


ABBILDUNG 78. Verwendete Produkte des Prozesses *System-Komponenten bearbeiten*

5.1 Prozessverfeinerung

Der Prozess *System-Komponenten bearbeiten* unterteilt sich in zwei Prozesse (siehe Abbildung 79). Ausgehend vom UML-Entwurf und den Komponenten-Anforderungen wird im Prozess *Quellkode bearbeiten* der *Komponenten-Kode* für die einzelnen Komponenten entwickelt. Die *Entwicklungsrichtlinien* geben Anleitungen zur Durchführung des Prozesses. Anschließend wird der Komponenten-Kode übersetzt. Teil des Prozesses *Quellkode bearbeiten* ist außerdem die Verifikation des *Quellkodes* gegen den Entwurf und die *Entwicklungsrichtlinien.* Die Ergebnisse der Verifikation werden im Kapitel *K-Verifikation* der *Software-Komponenten-Dokumentation* dokumentiert. Im Prozess *Ausführbare Komponente bearbeiten* werden für jede Komponente *Test-Rahmen* entsprechend der im Dokument E-Validation festgehaltenen Testpläne entwickelt. Der übersetzte *Komponenten-Kode* und ein übersetzter *Test-Rahmen* werden zusammengebunden. Das *ausführbare Binary (*als Teil des *ausführbaren Komponenten-Kodes)* wird ausgeführt und Testausgaben werden im Dokument E-Validation protokolliert. Bei auftretenden Fehlverhalten werden die Fehler in der gesamten System-Dokumentation (d. h. Problemlösungsdokumentation, Software-System-Dokumentation und *Software-Komponenten-Dokumentation*) gesucht und behoben.

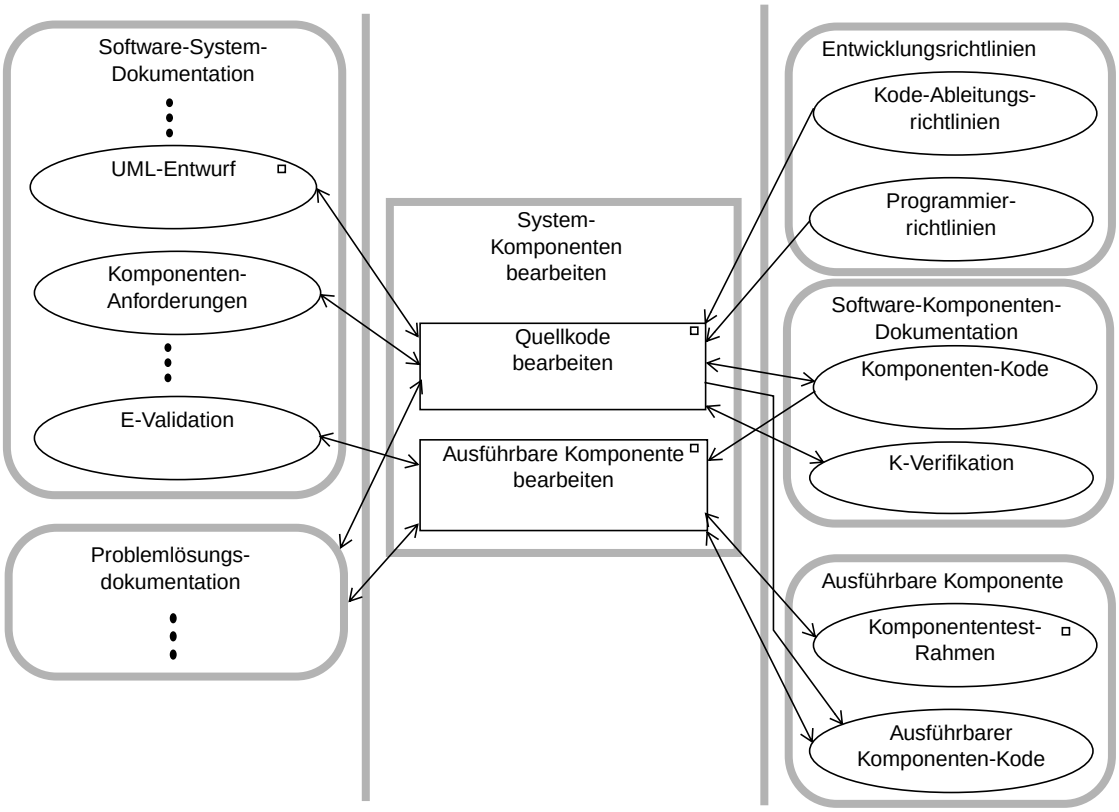


ABBILDUNG 79. Verfeinerung des Prozesses *System-Komponenten bearbeiten*

5.2 Produkte

Im folgenden wird eine Übersicht über die im Prozess *System-Komponenten bearbeiten* verwendeten Produkte gegeben. Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte der Dokumente.

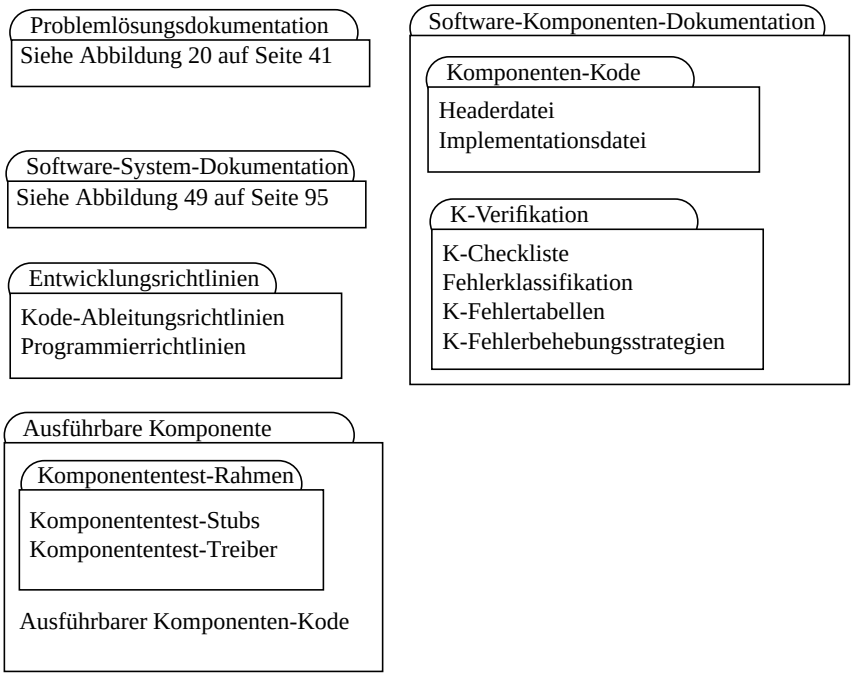


ABBILDUNG 80. Produktübersicht für den Prozess *System-Komponenten bearbeiten*

Komponenten-Kode

Der *Komponenten-Kode* beschreibt die Realisierung einer Komponente in einer konkreten Programmiersprache. Zur Unterstützung der Verständlichkeit und Wartbarkeit des *Kodes* sollten pro Komponente zwei *Kode-Dateien* angelegt werden: 1. eine *Headerdatei*, die die Schnittstelle der Komponente zu anderen Komponenten beschreibt und 2. eine *Implementationsdatei*, die die Realisierung der Komponente enthält.

Die Verwendung eines CASE-Werzeugs (wie z. B. StP/UML) erlaubt eine Generierung der Klassenheader mit Attributen und Methoden, so wie sie in den E-Klassendiagrammen und den Class-Tables definiert sind. In dem generierten

Komponenten-Kode werden außerdem angelegte informelle Beschreibungen als Kommentierung übernommen.

Abbildung 82 zeigt einen Ausschnitt aus der *Header*- und der *Implementationsdatei* für die Klasse “LightController” des Teilsystems “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *Komponenten-Kode*

Die Komponenten des Gebäudeautomationssystems wurden in der Sprache C++ programmiert. Die Klassenheader ließen sich mit Hilfe des verwendeten CASE-Werkzeugs StP/UML generieren. Zur Generierung verwendet StP/UML die Informationen aus den E-Klassendiagrammen und den Class-Tables. Die Informationen aus den E-Zustands- und Aktivitätsdiagrammen bleiben ungenutzt.

Im Gebäudeautomationsprojekt stellte sich eine Generierung der Klassenheader und das Übernehmen von informellen Beschreibungen als effizienter heraus, als eine vollständige manuelle Implementierung der Komponenten. Allerdings ließ sich generierter und anschließend bearbeiteter *Kode* nicht neu generieren, ohne die durchgeführten Änderungen am *Kode* zu verlieren.

Das folgende Beispiel zeigt einen Ausschnitt aus der *Headerdatei* der Klasse “LightController”. Die Headerdatei ließ sich mit Hilfe von StP/UML fast vollständig generieren.

```
//  Class: LightController

// Description:
// Ein LightController-Objekt dient einerseits als aktiver Melder,
// andererseits als Steurungseinheit fuer die Lichtanlage im Flur. Als Melder
// gibt er Zustandsaenderungen der Lichtanlage (das Licht wurde durch den
// Benutzer manuell ein-/ausgeschaltet) an das LightControlUnit-Objekt weiter.
// Als Steuerungseinheit verfuegt die Klasse ueber Methoden zum Ein- bzw.
// Ausschalten der Lichtanlage.

class LightController : public ScheduledFct
{
// stp class members
public:

//  Operation: lightControl
// Description:
// Die Methode lightControl() ruft in Abhaengigkeit von ihrem Parameter
// LichtStatus die Methoden SwitchOn() bzw. SwitchOff() eines Objekts
// der Klasse Light auf, um die Lichtanlage des Flurabschnitts entsprechend
// anzusteuern.

    void lightControl(zLightStatusType LightStatus);

// Operation: callBack
// Description:
// Diese Methode wird von der Klasse ScheduledFct geerbt.
// Sie dient der Abwicklung der zeitabhaengigen Ablaeufe.
    void callBack(void);

...

private:
// Attribute: zLCState
// Description:
// Die Enumeration definiert die gueltigen Zustaende
// eines Objekts der Klasse LightController
  enum {cStatusLichtAus, cStatusLichtAn} zLCState;
  zLightStatusType zLightStatus;
```

ABBILDUNG 81. Ausschnitt aus der Headerdatei der Klasse “LightController”

Das nächste Beispiel zeigt einen Ausschnitt aus der *Implementierungsdatei* der Klasse “LightController”. Die Realisierung der Methode “lightControl” wird beschrieben. Der Header der Methode konnte generiert werden. Der Methodenrumpf musste manuell eingefügt werden.

```
// stp operation 337::340
void LightController::lightControl(zLightStatusType LightStatus) {

  switch(zLCState) {

  case cStatusLichtAus:
     if (LightStatus == cLightOn) {
        // Zustandwechsel
     zLCState=cStatusLichtAn;
        // Licht einschalten
     lightPtr->SwitchOn();
     }
     break;
   // -- Ende des case-Block cStatusLichtAus --

  case cStatusLichtAn:
     if (LightStatus == cLightOff) {
        // Zustandwechsel
     zLCState=cStatusLichtAus;
        // Licht ausschalten
     lightPtr->SwitchOff();
     }
   break;
   // -- Ende von case-Block cStatusLichtAn --

  default :
     break;
  }
}// ------ lightControl(zLightStatusType LightStatus) ------
// end stp operation
```

ABBILDUNG 82. Ausschnitt aus der Implementationsdatei der Klasse “LightController”

K-Verifikation

Das Dokument *K-Verifikation* enthält Hinweise zu und Ergebnisse von durchgeführten Verifikationsaktivitäten auf Implementierungsebene.

Im Folgenden wird sich auf eine Inspektion mit Checklisten als Verifikationsaktivität eingeschränkt. Das Dokument *K-Verifikation* umfasst dann: 1. eine *Checkliste*, 2. eine *Fehlerklassifikation*, 3. eine *Liste der gefundenen Fehler* und 4. eine *Beschreibung der Fehlerbehebungsstrategien*.

K-Checkliste

Die *Checkliste* unterstützt das Lesen des zu verifizierenden Komponenten-Kodes und erleichtert das Finden von Problemen im Dokument.

Bei der Überprüfung des Komponenten-Kode müssen - wie bei der Verifikation der System-Anforderungen und des Entwurfs - drei Aspekte überprüft werden: 1. Sind die Dokumentationsrichtlinien erfüllt (d. h., ist das Dokument syntaktisch korrekt), 2. Ist die Qualität des Dokuments ausreichend (d. h., ist das Dokument semantisch korrekt) und 3. Ist das Dokument mit dem Vorgängerdokument konsistent. In der *Checkliste* werden zu jedem Aspekt Fragen gestellt. Werden Fragen in der *Checkliste* mit Nein beantwortet, dann könnte es sich um einen Fehler handeln.

Abbildung 83 zeigt einen Ausschnitt aus einer *Checkliste* zur Überprüfung des Komponenten-Kodes.

Beispiel *K-Checkliste*

Überprüfe zu jeder Klasse die .hh und .cc-Datei auf:

Einhaltung der Dokumentationsrichtlinien:

1. Wurden die Entwicklungsrichtlinien eingehalten?

   Hier ist insbesondere die Einhaltung der Programmierrichtlinien zu überprüfen (Namenskonventionen, Vermeidung von bestimmten Anweisungstypen, Gestaltung und Dokumentation des Kodes, usw.).

Interne Konsistenz des Kodes:

1. *Wenn einer Variablen ein Wert zugewiesen wird (die Variable wird definiert):* Wird die Variable im weiteren verwendet (referenziert)?

   Zwei Arten von Anomalien können gefunden werden: DD (Definiert, Definiert) und DU (Definiert, Undefiniert). Diese Anomalien sind Fehler, die nicht direkt zu einem Fehlverhalten des Systems führen.

Konsistenz zum Entwurf:

1. Sind alle Attribute aus den Class-Tables im Kode enthalten?
2. Sind alle Methoden mit Parametern und Rückgabetypen aus den Class-Tables im Kode enthalten?

ABBILDUNG 83. Ausschnitt aus der K-Checkliste (Komponenten-Kode)

Fehlerklassifikation

Die aufgedeckten *Fehler* werden in einer Fehlertabelle aufgelistet. Dabei wird jeder *Fehler* einer *Fehlerklasse* zugeordnet. Die *Fehlerklassifikation* ist identisch zur *Fehlerklassifikation* im Dokument A-Verifikation der System-Anforderungen

(siehe Abbildung 32, „Ausschnitt aus einer Fehlerklassifikation - Spezifische Fehlerklassen,“ auf Seite 60).

K-Fehlertabellen

In den *K-Fehlertabellen* werden alle *Fehler*, die im Produkt Software-Komponenten-Dokumentation gefunden wurden, aufgelistet. Die *Tabelle* umfasst einerseits Fehler, die während der Inspektion des Komponenten-Kodes gefunden wurden. Andererseits enthält sie Fehler, die Ursachen von Fehlverhalten sind. Die Fehlverhalten wurden beim Test ausführbarer Produkte entdeckt und in der *K-Fehlertabelle* dokumentiert. Der Prozess, während dessen ein *Fehler* gefunden wurde, ist in der *Fehlertabelle* vermerkt (z. B. “komkodrevvorb” für “Komponenten-Kode Review vorbereiten” oder “systestdur” für “Systemtest durchführen”). Die *Fehler* werden dabei mit Hilfe des Produkts und des Teilsystems, in dem sie aufgetreten sind, gekennzeichnet (*Produkt*-*Teilsystem*-FNr-*Nummer*). Zu jedem *Fehler* wird die vom *Fehler* betroffene Datei angegeben.

Abbildung 84 zeigt eine *Tabelle zur Dokumentation gefundener Fehler.*

Beispiel *K-Fehlertabellen*

Die *K-Fehlertabelle* dokumentiert einen *Fehler* im Teilsystem “LichtregelungFlur” im Komponenten-Kode, der beim Lesen des Kodes gefunden wurde. Der *Fehler* befindet sich in der Datei “LightController.cc”, die die Implementierung der Klasse “LightController” enthält. Die Fehlertabelle enthält außerdem Fehler im Kode, die Ursachen für Fehlverhalten sind. Die Fehlverhalten wurden bei verschiedenen Tests (KV-Komponentenvalidation und ST-Systemtest) gefunden und auf Fehler im Kode zurückgeführt. Ein Verweis auf die Fehler wird in der Tabelle zur Dokumentation der Fehlverhalten ergänzt.

Der Fehler K-LF-FNr3 ist ein Folgefehler eines Fehlers in den System-Anforderungen. Der Fehler muss im Kode, im Entwurf und in den Anforderungen behoben werden. Die Behebung der Fehler wird in den Tabellen zur Dokumentation der Fehlerbehebungsstrategien in den Verifikationskapiteln dokumentiert.

F-Nr	betroffene Datei	allgemeine Fehlerbeschreibung	Fehlerklasse	Prozess während Fehler gefunden wurde
K-LF-FNr1	LightCont-roller.cc	Methode lightControl definiert, aber nicht imple-mentiert.	U	komkodrevvorb
K-LF-FNr2	LightCont-roller.cc	Die Abfrage der aktuellen Zeit wurde vergessen.	U	komtestdur
K-LF-FNr3	LightCont-roller.cc	Beim Zustandsübergang wird SwitchOff() anstelle SwitchOn() aufgerufen	F	systestdur

ABBILDUNG 84. Ausschnitt aus der K-Fehlertabelle für das Teilsystem “LichtregelungFlur”

K-Fehlerbehebungsstrategien

Die Behebung der Fehler aus der K-Fehlertabelle wird in den Tabellen zur Dokumentation der Fehlerbehebungsstrategie in den Verifikationskapiteln dokumentiert. Beispielsweise wird die Behebung des Fehlers K-LF-FNr-3 in den Verifikationskapiteln der System-Anforderungen, des Entwurfs und des Kodes dokumentiert. In der *Tabelle zur Dokumentation der Fehlerbehebungsstrategien* wird verdeutlicht, welche Fehler aus der Fehlerliste behoben wurden und wie dies geschehen ist. Die *Tabelle* gibt damit Aufschluss über den Stand der Rework-Aktivitäten. Anhand der Tabelle kann außerdem entschieden werden, ob eine erneute Verifikation des geänderten Komponenten-Kodes notwendig ist.

Abbildung 85 zeigt einen Ausschnitt aus einer *Tabelle zur Dokumentation der Fehlerbehebungsstrategie* für das Teilsystem “LichtregelungFlur”.

Beispiel *K-Fehlerbehebungsstrategien*

In der *Tabelle* werden alle Fehler aus der Fehlertabelle aufgeführt. Die Beschreibung der Strategien zur Behebung der Fehler erlaubt es, nachzuvollziehen, was sich im Komponenten-Kode geändert hat.

Referenz auf FNr	Strategie zur Behebung des Fehlers	Behebung des Fehlers durchgeführt? (J/N)
K-LF-FNr1	Methode requestLightStatus() implementiert.	J
K-LF-FNr2	Abfrage der aktuellen Zeit wurde integriert	J
K-LF-FNr3	Aufruf beim Zustandsübergang wurde von SwitchOff() in SwitchOn() geändert.	J

ABBILDUNG 85. Ausschnitt aus der Tabelle zur Dokumentation der Fehlerbehebungsstrategien - Teilsystem Lichtregelung Flur

Entwicklungsrichtlinien

Die *Entwicklungsrichtlinien* bestehen aus den *Kode-Ableitungsrichtlinien* und den *Programmierrichtlinien*.

Kode-Ableitungsrichtlinien

Die *Kode-Ableitungsrichtlinien* leiten den Kodierer an, die Information der Zustandsdiagramme aus dem UML-Entwurf zu implementieren. Der Kodierer wird dabei Schritt für Schritt bei der Umsetzung aller Teile eines Zustandsdiagramms angeleitet. Zum besseren Verständnis der *Richtlinien* werden Beispiele gegeben.

Bei der Implementierung unter Verwendung der *Richtlinien* erfolgt eine Schachtelung von Methoden analog zum Aufbau der Zustandsdiagramme. Jeder Superstate sowie die parallelen Substates werden durch eine eigene Methode realisiert. Die Hierarchie der Zustandsdiagramme und die Trennung von parallelen Substates als voneinander unabhängige Teile bleiben in der Implementierung sichtbar.

Durch die *Vorgaben* und *Empfehlungen* soll insbesondere die Verfolgbarkeit vom Entwurf zum Kode gewährleistet werden. Andere Ziele, wie beispielsweise eine effiziente Implementierung der Komponenten, haben niedrigere Priorität.

Wenn derartige *Kode-Ableitungsrichtlinien* an die konkreten Charakteristika eines Software-Systems angepasst werden (z. B. an die Verwendung eines Schedulers), kann der Implementierungsprozess sehr detailliert angeleitet werden.

Abbildung 86 zeigt einen Ausschnitt aus den *Kode-Ableitungsrichtlinien*, die im Gebäudeautomationsprojekt verwendet wurden.

Beispiel *Kode-Ableitungsrichtlinien*

Die im Projekt verwendeten *Kode-Ableitungsrichtlinien* waren wie folgt aufgebaut:

- ***Problem:*** zeigt kurz, worum es in der Richtlinie geht.
- ***Richtlinien*** sind aus dem Problem abgeleitet und verbindlich für den Implementierer.
- ***Bemerkungen*** zur Problematik oder zur Verwendung der Richtlinien.
- ***Empfehlungen*** sollten nach Möglichkeit berücksichtigt werden.
- ***Beispiel:*** kurzes Kodebeispiel zur Verdeutlichung der Richtlinien.

Es folgt ein Ausschnitt aus den *Richtlinien*, der beschreibt, wie man eine Zustandskonfiguration eines Zustandsautomaten implementiert.

Die Zustandskonfiguration ohne Verwendung von IN

Problem

Ein Objekt muss sich merken, in welcher Zustandskonfiguration sich sein Zustandsautomat momentan befindet. Dabei sollen an das Zustandsdiagramm angelehnte Zustandsbezeichnungen verwendet werden.

Richtlinien

Zur Darstellung der Zustandskonfiguration einer Klasse wird eine Reihe von Attributen definiert. Dies geschieht nach folgenden Regeln, wobei auf die rekursive Anwendbarkeit auf Substates zu achten ist:

- Ist ein Zustand vom Typ OR, so wird für ihn ein Attribut definiert. Dieses stellt einen Aufzählungstyp (enum) dar, dessen Werte durch die möglichen Substates gegeben sind.
- Ist der Zustand vom Typ AND, so erhält er kein eigenes Attribut. Für jeden seiner parallelen Substates wird ein Attribut definiert.
- Zustände vom Typ BASIC finden sich als Attributwerte ihres Superstates wieder.

Bemerkung

Da eine geschachtelte Definition von Aufzählungstypen nicht möglich ist, wurde hier auf die Darstellung der zusammengesetzten Zustandskonfiguration durch mehrere Attribute ausgewichen. Da C++ für die Werte einer Aufzählung Konstanten definiert, ist auf die Eindeutigkeit der Attributwerte zu achten.

Empfehlungen

Attributbezeichnern sollte „state“ vorangestellt werden.
Um Attributwerte eindeutig zu machen, sollten sinnvolle Ergänzungen der Bezeichner durch Teile des Namens des Superstates gemacht werden.

Beispiel

Im Ampelbeispiel ergeben sich folgende Attribute:

```
// Root-Zustand Ampel ist vom Typ OR -> 1 Attribut
enum {cBlinkend, cNormalBetrieb} stateAmpel;
// Normalbetrieb ist vom Typ OR -> 1 Attribut
enum {cOwGruen, cNsGruen} stateNormalbetrieb;

// OWGruen ist vom Typ AND -> 2 Attribute für die 2 Substates
enum {cNsBereit, cNsSchuss} stateKameraNS;
enum {cNsKeinAuto, cNsEinAuto, cNsZweiAutos} stateZaehlerNS;

// NSGruen ist vom Typ AND -> 2 Attribute für die 2 Substates
enum {cOwBereit, cOwSchuss} stateKameraOW;
enum {cOwKeinAuto, cOwEinAuto, cOwZweiAutos} stateZaehlerOW;
```

ABBILDUNG 86. Ausschnitt aus den Kode-Ableitungsrichtlinien

Programmierrichtlinien

Im Dokument *Programmierrichtlinien* werden *Richtlinien* gegeben, die zu einer einheitlichen Gestaltung des Quellkodes führen (z. B. durch Vorgaben zur Klammerung oder zur Gestaltung von Headern von Dateien). Die *Richtlinien* gewährleisten eine einheitliche Gestaltung von Komponenten-Kode, der von verschiedenen Implementierern entwickelt wurde. Sie erleichtern ein Verständnis

des Quellkodes beispielsweise für den Wartungsingenieur oder Verifizierer. Die *Programmierrichtlinien* enthalten außerdem *Regeln*, die die Mächtigkeit der verwendeten Programmiersprache einschränken. Das ist insbesondere bei mächtigen Programmiersprachen wie C++ sinnvoll.

Abbildung 87 zeigt einen Ausschnitt aus den *Programmierrichtlinien* des Gebäudeautomationsprojekts.

Beispiel *Programmierrichtlinien*

Aufbau der Header-Datei

- Die Header-Datei beginnt immer mit den beiden Statements:

  #ifndef _KLASSE_H

  #define _KLASSE_H

  und endet mit der Zeile:

  #endif

  Dabei ist KLASSE durch den jeweiligen Klassennamen zu ersetzen.
- Im Kopf der Datei steht (als Kommentar) der Autor, das Erzeugungsdatum und die Dateihistorie. Die Einträge werden von RCS automatisch generiert.
- Danach kommen alle include-Statements, die benötigt werden.

Aufbau der Code-Datei

- Im Kopf der Datei steht wie bei der Header-Datei (als Kommentar) der Autor, das Erzeugungsdatum und die Dateihistorie.
- Danach wird die Datei <Klasse.hh> inkludiert:

  #include <Klasse.hh>

  Dabei ist Klasse durch den jeweiligen Klassennamen zu ersetzen.
- Daran schließt sich der Quellcode für die einzelnen Methoden an.

Code

- Alle anderen Methoden und alle Attribute sind *private*.
- Globale Variablen sind nicht erlaubt.
- Auf das Konstrukt “extern” ist zu verzichten.
- Variablen sollen nur am Anfang eines Blocks deklariert und initialisiert werden.
- In jeder Zeile steht nur eine Anweisung.
- Es sollen keine *type casts* verwendet werden.

ABBILDUNG 87. Ausschnitt aus Programmierrichtlinien für C++

Ausführbare Komponente

Die *Ausführbare Komponente* besteht aus dem *Komponententest-Rahmen* und dem *Ausführbaren Komponenten-Kode*.

Komponententest-Rahmen

Der *Komponententest-Rahmen* beinhaltet den Quellkode der Treiber und Stubs, die zum Test der Komponenten benötigt werden. Die Funktionalität der Test-Treiber und Test-Stubs ist in den Testplänen für den Komponententest im Kapitel E-Validation beschrieben.

Komponententest-Treiber

Der *Komponententest-Treiber* ist ein Modul (z.B. Testfall.cc) mit der Funktion main, in der Methoden der zu testenden Klasse aufgerufen werden. Welche Objekte im *Komponententest-Treiber* angelegt und welche Methoden der zu testenden Komponente aufgerufen werden müssen, ist in den Testplänen vermerkt. Relevante Testergebnisse werden vom *Test-Treiber* nach cout geschrieben.

Abbildung 88 zeigt einen Ausschnitt aus einem *Test-Treiber* für die Klasse “Light-Controller” des Teilsystems “LichtRegelungFlur” aus dem Gebäudeautomationssystem.

Beispiel *Komponententest-Treiber*

```
#include <Testfall.hh>

void main()
{
 cout << “-------------------------------------------------” << endl;
 cout << “Ausfuehrung von Testfall 1 (KompTest1) zur Klasse LightController”
      << endl;
 cout << “-------------------------------------------------” << endl << endl;
 cout << “Verwendete Stubs:” << endl << endl;
 cout << “Light” << endl;
 cout << “ ->IsOn():zLightStatusType” << endl;
 cout << “ ->SwitchOn():void” << endl;
 cout << “ ->SwitchOff():void” << endl << endl;
 cout << “LightControlUnit” << endl;
 cout << “ ->changedLightStatus()::void” << endl << endl;
 cout << “-------------------------------------------------” << endl;
 cout << “------------------ Testlauf beginnt ---------------” << endl;
 cout << “-------------------------------------------------” << endl << endl;

 int sectionNr=1;
 int roomNr=0;
 zLightStatusType lightStatus;

 cout << “Schritt 1: Aufruf des LightController-Konstruktors” << endl;
 LightController* lightControllerPtr = new LightController(sectionNr, roomNr);
 // Erzeugen des LightControlUnit-Stub-Objekts
 LightControlUnit* lightControlUnitPtr = new LightControlUnit(lightControllerPtr);
 cout << endl;

 cout << "Schritt 2: Aufruf der Methode setLightControlUnit" <<
   endl << "(LightControlUnit*)" << endl;
 lightControllerPtr->setLightControlUnit(lightControlUnitPtr);
 cout << endl;

 cout << "Schritt 3: Aufruf der Methode callBack()" << endl;
 lightControllerPtr->callBack();
 cout << endl;
```

ABBILDUNG 88. Ausschnitt aus einem Test-Treiber zum Test der Komponente “LightController”

Komponententest-Stubs

Die *Komponententest-Stubs* sind Dummies von Klassen, die durch die zu testende Klasse aufgerufen werden. Die Funktionalität (Methoden, Rückgabewerte und Ausgaben), die die *Komponententest-Stubs* bieten müssen, sind in den Testplänen zum Komponententest (siehe Kapitel E-Validation) dokumentiert. Relevante Testergebnisse werden von den *Komponententest-Stubs* nach cout geschrieben.

Abbildung 89 zeigt einen Ausschnitt aus einem *Test-Stub* für die Klasse “Light-Controller” des Teilsystems “LichtRegelungFlur” aus dem Gebäudeautomations-system.

Beispiel *Komponententest-Stub*

```
class Light // : public Actuator
{
// stp class members

public:
// Operation: SwitchOn
// Description:
// Diese Methode schaltet das Licht an.
    void SwitchOn(void) {

   counterSwitchOn++;

   cout << counterSwitchOn << “. Aufruf: Methode Light->SwitchOn()” << endl;

   switch(counterSwitchOn) {

   case 1:
     cout << “>> Das Licht wurde eingeschaltet” << endl;
     break;

   default:
     cout << “Unzulaessiger Aufruf!” << endl;
     break;
   }
}
// Konstruktor
  Light(int SectionNr, int RoomNr, char* ActuatorName) {
   counterSwitchOn = 0;

   cout << "Light(" << SectionNr << ", " << RoomNr << ", " << ActuatorName
        << ")" << endl;
  }
...
protected:
  int counterSwitchOn;
...
}
```

ABBILDUNG 89. Ausschnitt aus einem Stub für den Test der Komponente “LightController)

Ausführbarer Komponenten-Kode

Der *Ausführbare Komponenten-Kode* enthält für jede Komponente den übersetzten Komponenten-Kode und die zum Test benötigten übersetzten Test-Rahmen. Außerdem umfasst es für jede Komponente mehrere lauffähige Binarys entsprechend der Anzahl der durchzuführenden Tests. Pro Test wird der übersetzte Komponenten-Kode mit dem passenden übersetzten Test-Rahmen zusammengebunden.

5.3 Rollen

Im Prozess *System-Komponenten bearbeiten* sind die Rollen *Kodierer*, *Verifizierer*, *Validierer*, *Anforderungsingenieur*, *Entwurfsingenieur*, und *Qualitätsmanager* involviert. Ihre Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 90. Folgende wesentliche Tätigkeiten sind im Prozess *System-Komponenten bearbeiten* mit den Rollen verbunden:

- ***Kodierer.*** Der Kodierer implementiert die System-Komponenten, nimmt an der Verifikation des *Komponenten-Kodes* teil und ist für die Fehlerbehebung in den ihm zugeordneten Dokumenten verantwortlich.
- ***Verifizierer.*** Die Aufgaben des Verifizierers sind, die Inspektion des Quellkodes zu organisieren, diese in Kooperation mit dem Kodierer durchzuführen und die Ergebnisse zu dokumentieren.
- ***Validierer.*** Die Aufgaben des Validierers im Prozess *System-Komponenten bearbeiten* sind, *Komponententest-Rahmen* zu erstellen, den Komponententest durchzuführen und die Ergebnisse zu dokumentieren.
- ***Anforderungsingenieur.*** Die Aufgabe des Anforderungsingenieurs ist, bei der Verifikation gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben. Hierfür kann er den Kunden kontaktieren, sofern die *Problembeschreibung* oder die *Informellen B-Anforderungen* betroffen sind.
- ***Entwurfsingenieur.*** Die Aufgabe des Entwurfsingenieurs ist, bei der Verifikation gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Qualitätsmanager.*** Die Aufgabe des Qualitätsmanagers ist, entsprechend den längerfristigen Bedürfnissen und Erwartungen einer Software-Entwicklungsorganisation Wissen/Erfahrung in geeigneter Form zu verwalten, einzelnen Projekten auf Anfrage zur Wiederverwendung zur Verfügung zu stellen und kontinuierlich aufgrund gemachter Projekterfahrungen zu verbessern. Wissen/Erfahrungen kann vom Qualitätsmanager im Prozess *System-Komponenten bearbeiten* z. B. in Form von Checklisten oder Fehlerklassifikationen zur Verfü-

gung gestellt werden. Diese sind zum Teil Bestandteil von Produkten, für die andere Rollen verantwortlich sind. Eigenverantwortlich ist der Qualitätsmanager hier für das Produkt *Entwicklungsrichtlinien*.

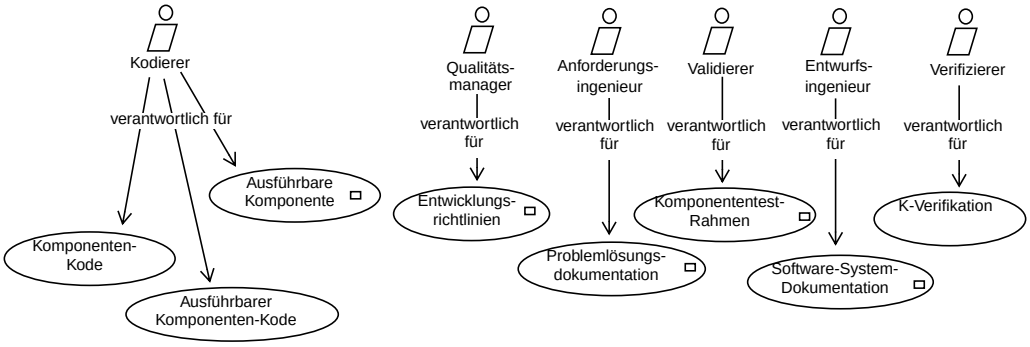


ABBILDUNG 90. Ressourcen und Produkte des Prozesses *System-Komponenten bearbeiten*

5.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 91 dargestellt.

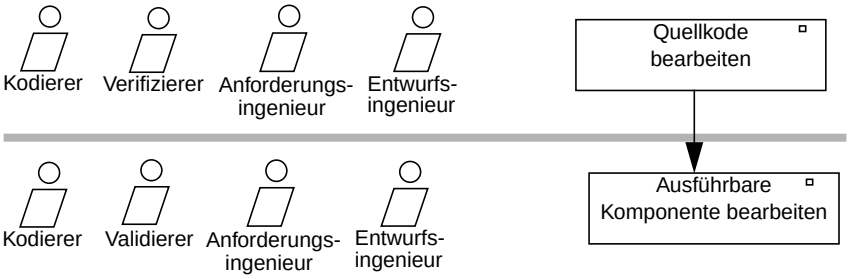


ABBILDUNG 91. Kontrollfluss für den Prozess *System-Komponenten bearbeiten*

5.5 Prozess “Quellkode bearbeiten”

Im Prozess *Quellkode bearbeiten* wird aus den E-Klassendiagrammen, den Class-Tables und den E-Zustands- und Aktivitätsdiagrammen des UML-Entwurfs sowie aus den Komponenten-Anforderungen der Komponenten-Kode entwickelt. Die Implementierung kann für jede Komponente unabhängig durchgeführt werden. Die Entwicklungsrichtlinien leiten den Implementierungsprozess an und führen zu einer einheitlichen Gestaltung des Komponenten-Kodes. Nach der fertiggestellten Implementierung wird der Komponenten-Kode übersetzt. Der Komponenten-Kode wird gegen den UML-Entwurf, die Komponenten-Anforderungen und gegen die Entwicklungsrichtlinien verifiziert. Die Ergebnisse der Verifikation werden im Dokument K-Verifikation dokumentiert.

Abbildung 92 zeigt den Prozess *Quellkode bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

Folgende Produkte werden im Prozess *Quellkode bearbeiten* konsumiert:

- Entwicklungsrichtlinien. Das Dokument beschreibt Richtlinien für die Implementierung von Komponenten.

Folgende Produkte werden modifiziert:

- Komponenten-Kode. Das Dokument beschreibt den Quellkode der einzelnen Komponenten.
- Ausführbarer Komponenten-Kode. Dieses Produkt beinhaltet den übersetzten Komponenten-Kode.
- K-Verifikation. Das Dokument beschreibt Anleitungen für und Ergebnisse von Verifikationsaktivitäten auf Implementierungsebene.
- UML-Entwurf.
- Verfolgbarkeitsmatrix-Entwurf.
- Komponenten-Anforderungen.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.

- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

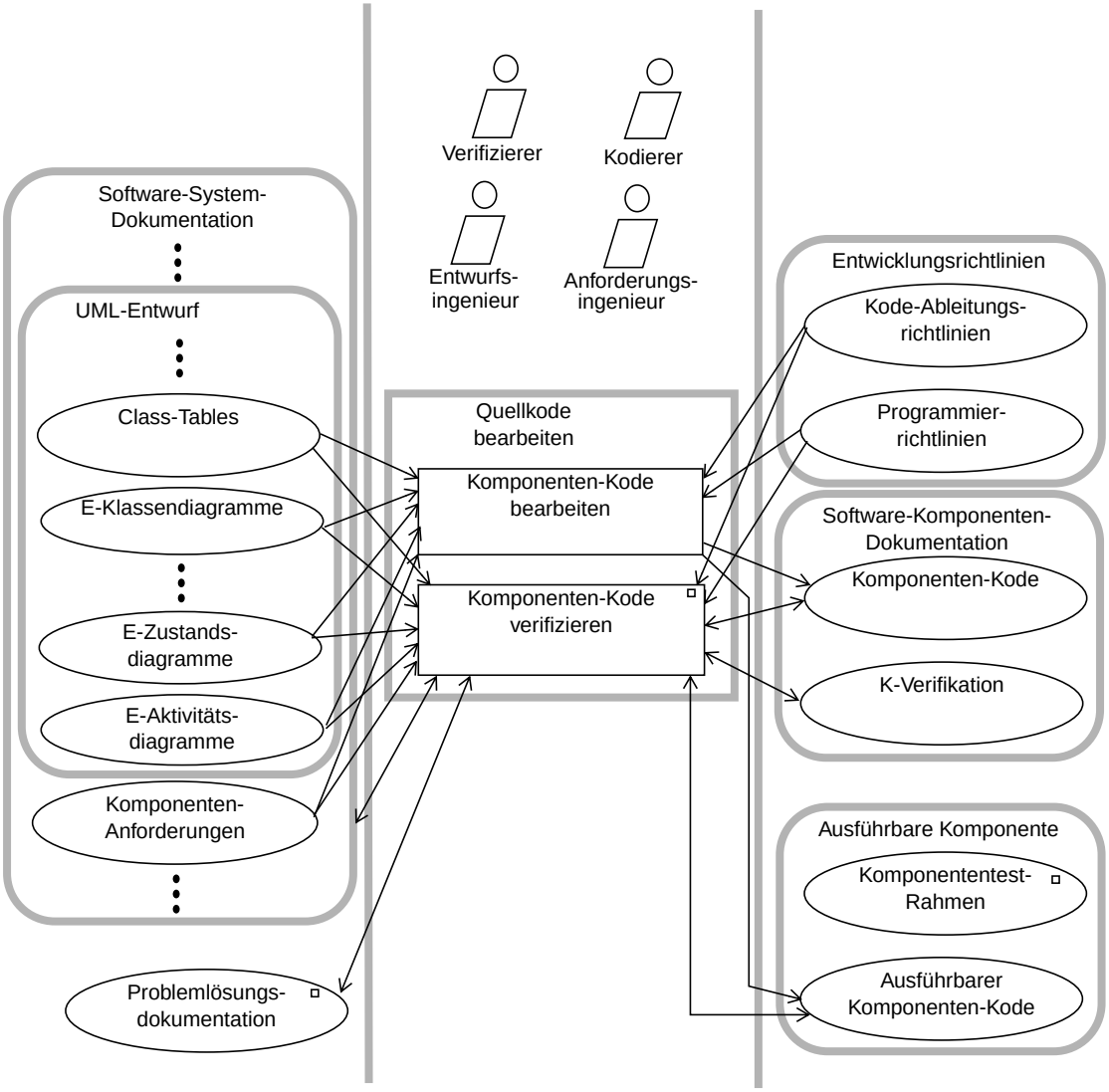


ABBILDUNG 92. Verwendete Produkte des Prozesses *Quellkode bearbeiten*

Prozess *"Komponenten-Kode bearbeiten"*

Im Prozess *Komponenten-Kode bearbeiten* werden die Komponenten aus dem UML-Entwurf und den Komponenten-Anforderungen implementiert.

Die Information aus den Klassendiagrammen und den Class-Tables lässt sich mit Hilfe von CASE-Werkzeugen meist automatisch in den Kode übernehmen. Wichtig hierbei ist, dass die vorgegebenen Programmierrichtlinien mit der Generierung des CASE-Werkzeugs abgestimmt sind und sich nicht widersprechen. Eine Nachbearbeitung des generierten Kodes insbesondere hinsichtlich Gestaltungsrichtlinien (z. B. Klammerung oder Namenkonventionen) sollte vermieden werden.

Für die manuelle Implementierung der Information aus den E-Zustands- und E-Aktivitätsdiagrammen bedarf es zusätzlicher Unterstützung. Die Kode-Ableitungsrichtlinien leiten den Prozess der Ableitung von Kode aus den E-Zustandsdiagrammen an.

Nach der fertiggestellten Implementierung wird der Komponenten-Kode übersetzt.

Folgende Produkte werden im Prozess *Komponenten-Kode bearbeiten* konsumiert:

- Entwicklungsrichtlinien.
- E-Klassendiagramme.
- Class-Tables.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.
- Komponenten-Anforderungen.

Folgende Produkte werden produziert:

- Komponenten-Kode.
- Ausführbarer Komponenten-Kode.

Prozess *"Komponenten-Kode verifizieren"*

Im Prozess *Komponenten-Kode verifizieren* wird der Komponenten-Kode gegen die E-Klassendiagramme, die Class-Tables, die E-Zustands- und E-Aktivitätsdiagramme sowie die Entwicklungsrichtlinien verifiziert. Anleitungen für und Ergebnisse der Verifikation werden im Dokument K-Verifikation dokumentiert.

Im Folgenden wird sich auf die Inspektion als Verifikationsaktivität eingeschränkt.

Ein Inspektionsprozess umfasst drei Teilprozesse: 1. Lesen des zu inspizierenden Dokuments (*Komponenten-Kode Review vorbereiten*), 2. Durchführen einer Inspektionssitzung zur Besprechung aufgedeckter Fehler (*Reviewsitzung Komponenten-Kode*) und 3. Überarbeiten des verifizierten Dokuments zur Behebung von Fehlern (*Rework Komponenten-Kode*).

Abbildung 93 zeigt den Prozess *Komponenten-Kode verifizieren* mit konsumierenden, produzierenden und modifizierenden Produkten.

Folgendes Produkt wird im Prozess *Komponenten-Kode verifizieren* konsumiert:

- Entwicklungsrichtlinien.

Folgende Produkte werden modifiziert:

- Komponenten-Kode.
- Ausführbarer Komponenten-Kode.
- K-Verifikation.
- UML-Entwurf.
- Verfolgbarkeitsmatrix-Entwurf.
- Komponenten-Anforderungen.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

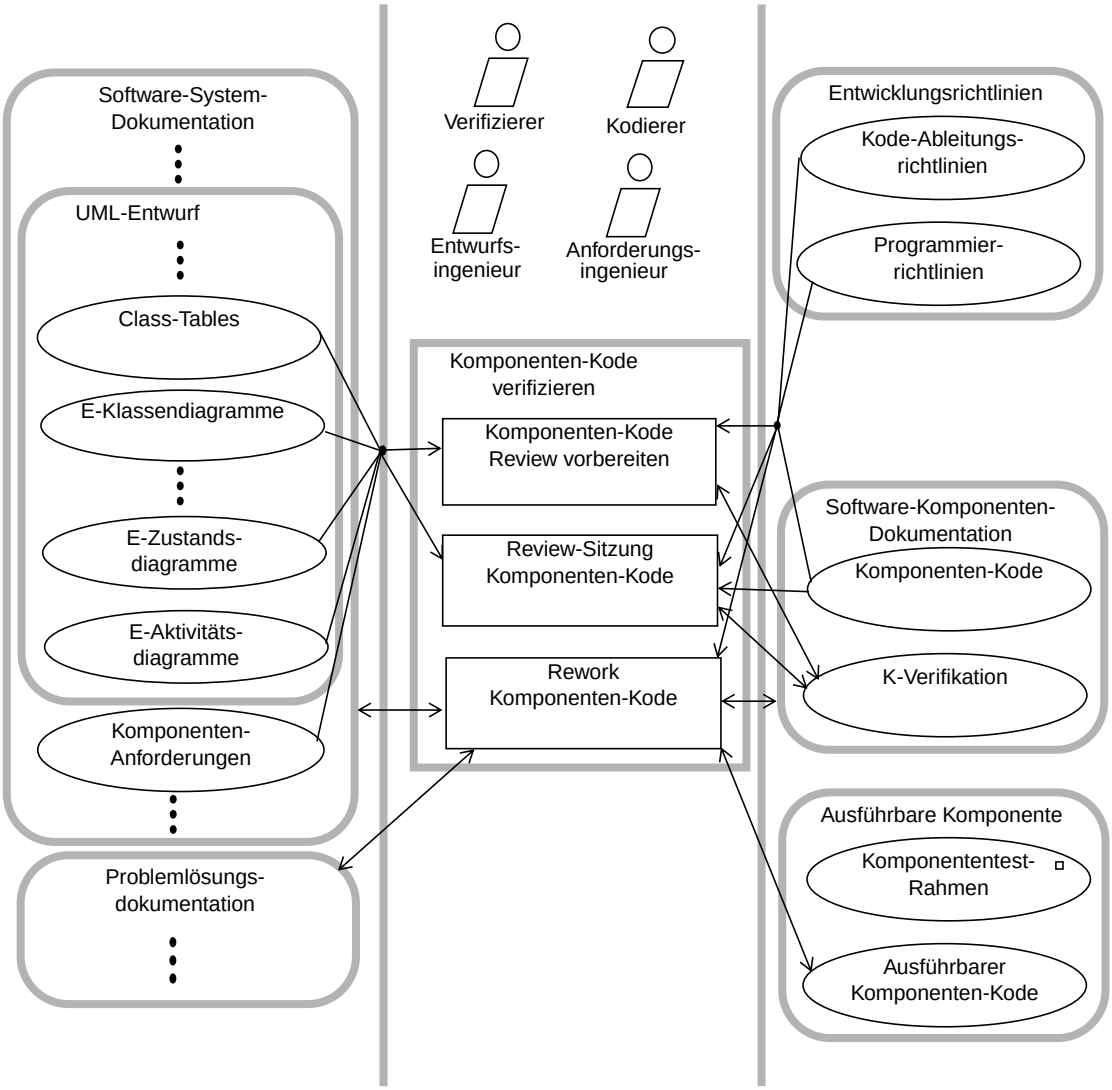


ABBILDUNG 93. Verwendete Produkte des Prozesses *Komponenten-Kode verifizieren*

Prozess “Komponenten-Kode Review vorbereiten”

Im Prozess *Komponenten-Kode Review vorbereiten* wird der Komponenten-Kode mit Hilfe der K-Checkliste, die im Dokument K-Verifikation dokumentiert ist, überprüft. Gelesen wird von den Verifizierern. Werden Fragen in der Checkliste mit “Nein” beantwortet, dann könnte es sich um einen Fehler handeln. Probleme im Dokument werden in der Fehlertabelle dokumentiert.

Ergebnis dieses Prozesses ist eine Liste mit möglichen Fehlern, die in sogenannten *K-Fehlertabellen* beschrieben wird. Jeder Fehler ist dabei einer Fehlerklasse zuzuordnen. Die potentielle Fehlerliste wird im Dokument K-Verifikation dokumentiert.

Im Prozess *Komponenten-Kode Review vorbereiten* werden folgende Produkte konsumiert:

- E-Klassendiagramme.
- Class-Tables.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.
- Komponenten-Anforderungen.
- Entwicklungsrichtlinien.
- Komponenten-Kode.

Im Prozess wird folgendes Produkt modifiziert:

- K-Verifikation.

Prozess “Reviewsitzung Komponenten-Kode”

Im Prozess *Reviewsitzung Komponenten-Kode* werden die beim Lesen gefundenen Fehler besprochen. An der Inspektionssitzung nehmen Kodierer und Verifizierer teil. Die Aufgabe der Verifizierer ist es nicht, die erkannten Fehler zu lösen, sondern lediglich, diese während der Sitzung in den Dokumenten aufzuzeigen. Die Verifizierer führen durch die Liste der Fehler. Ergebnis der Inspektionssitzung ist eine überarbeitete Fehlerliste, die im Dokument K-Verifikation festgehalten wird.

Im Prozess *Reviewsitzung Komponenten-Kode* werden folgende Produkte konsumiert:

- E-Klassendiagramme.

- Class-Tables.
- E-Zustandsdiagramme.
- E-Aktivitätsdiagramme.
- Komponenten-Anforderungen.
- Entwicklungsrichtlinien.
- Komponenten-Kode.

Im Prozess wird folgendes Produkt modifiziert:

- K-Verifikation.

Prozess “Rework Komponenten-Kode”

Im Prozess *Rework Komponenten-Kode* werden die Fehler aus der Fehlerliste behoben. Dabei werden die Strategie zur Behebung des Fehlers dokumentiert, um Änderungen an den Dokumenten zu verdeutlichen. Dies geschieht mit Hilfe der Tabelle zur Dokumentation der Fehlerbehebungsstrategien im Dokument K-Verifikation. Lässt sich ein Fehler auf einen Fehler in den System-Anforderungen oder im Entwurf zurückführen, wird der Fehler auch in den System-Anforderungen und im Entwurf behoben. Die Behebung wird in den Dokumenten A-Verifikation und E-Verifikation beschrieben.

Im Prozess *Rework Komponenten-Kode* wird folgendes Produkt konsumiert:

- Entwicklungsrichtlinien.

Folgende Produkte werden modifiziert:

- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.

- E-Validation.
- Komponenten-Kode.
- Ausführbarer Komponenten-Kode.
- K-Verifikation.

5.6 Prozess “Ausführbare Komponente bearbeiten”

Im Prozess *Ausführbare Komponente bearbeiten* werden Komponententest-Rahmen für den Test der Komponenten implementiert und übersetzt. Zur Durchführung eines Tests wird der übersetzte Komponenten-Kode mit dem zum Test gehörenden Test-Rahmen zusammengebunden. Das Ergebnis ist ein ausführbares Binary. Das ausführbare Binary wird ausgeführt und Testausgaben werden im Dokument E-Validation protokolliert. Die Testausgaben werden mit dem erwarteten Verhalten, das in den Testplänen beschrieben ist, verglichen und Fehlverhalten werden dokumentiert. Die Fehlverhalten werden auf Fehler in der System-Dokumentation zurückgeführt und behoben.

Abbildung 94 zeigt den Prozess *Ausführbare Komponente bearbeiten* mit konsumierenden, produzierenden und modifizierenden Produkten.

Folgende Produkte werden im Prozess *Ausführbare Komponente bearbeiten* modifiziert:

- Ausführbare Komponente.
- Komponenten-Kode.
- K-Verifikation.
- UML-Entwurf.
- Verfolgbarkeitsmatrix-Entwurf.
- Komponenten-Anforderungen.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.

- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

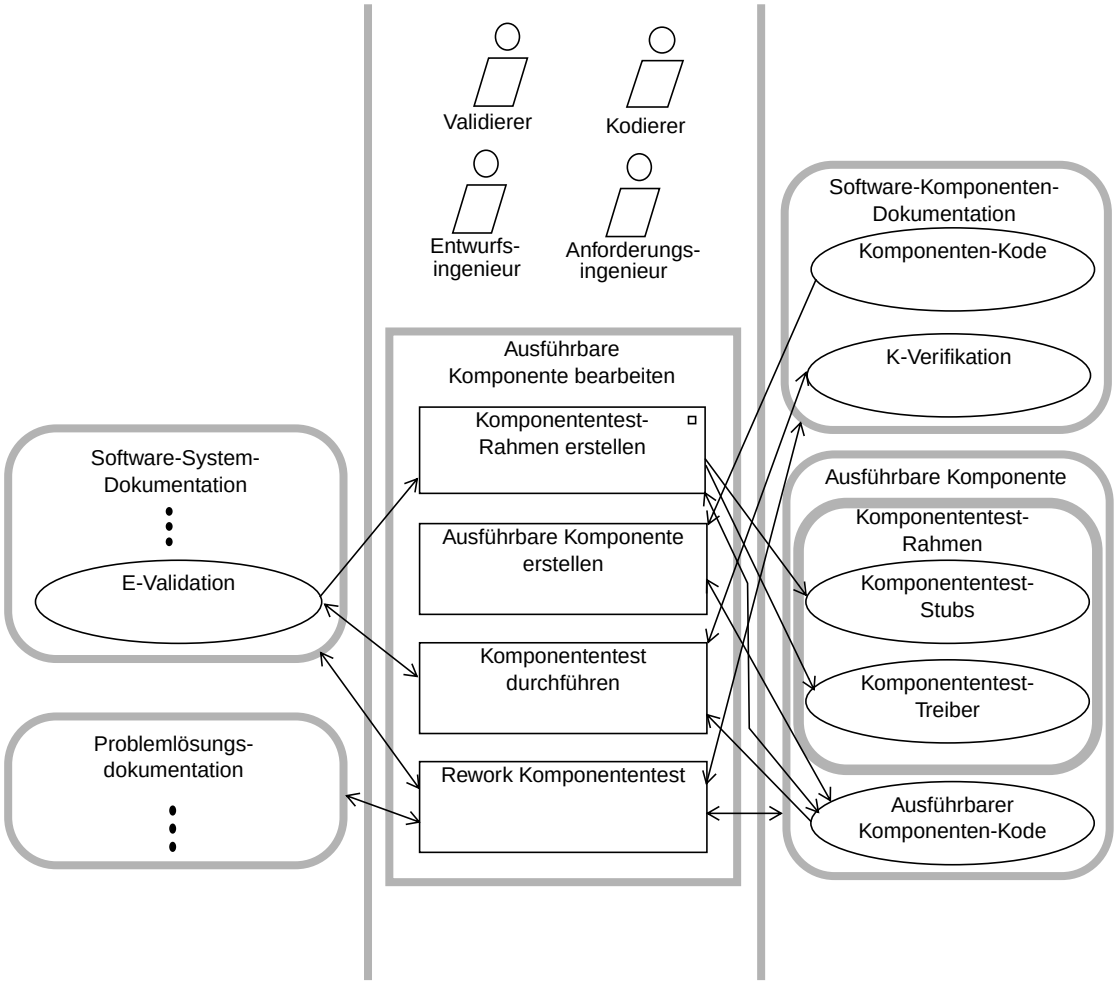


ABBILDUNG 94. Verwendete Produkte des Prozesses *Ausführbare Komponente bearbeiten*

Prozess *“Komponententest-Rahmen erstellen”*

Im Prozess *Komponententest-Rahmen erstellen* werden zum Test der Komponenten Test-Treiber und -Stubs implementiert und übersetzt. Die Testpläne für den Test der Komponenten (siehe Dokument E-Validation) legen fest, welche Funktionalität

die Test-Treiber und Test-Stubs bieten müssen. Der Quellkode der Test-Treiber und Test-Stubs wird im *Komponententest-Rahmen* abgelegt; die übersetzten Test-Treiber und Test-Stubs im Produkt *Ausführbarer Komponenten-Kode.*

Abbildung 95 zeigt den Prozess *Komponententest-Rahmen erstellen* mit konsumierenden, produzierenden und modifizierenden Produkten.

Folgende Produkte werden im Prozess *Komponententest-Rahmen erstellen* konsumiert:

- E-Validation.

Folgende Produkte werden produziert:

- Komponententest-Stubs.
- Komponententest-Treiber.

Folgendes Produkt wird modifiziert:

- Ausführbarer Komponenten-Kode.

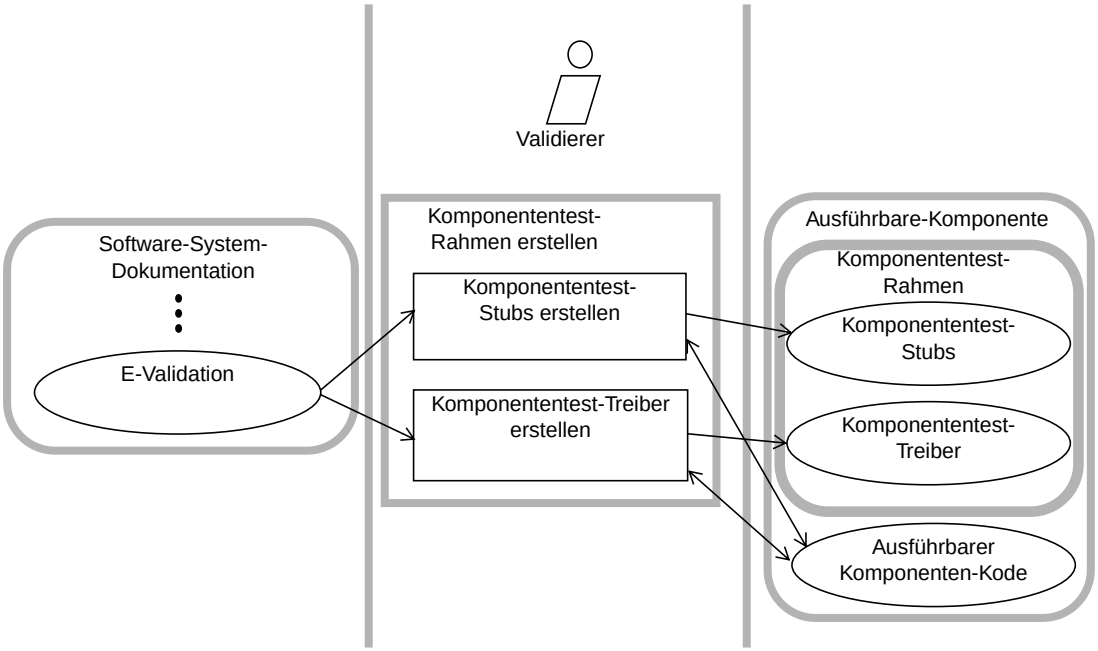


ABBILDUNG 95. Verwendete Produkte des Prozesses *Komponententest-Rahmen erstellen*

Prozess *“Ausführbare Komponente erstellen”*

Im Prozess *Ausführbare Komponente erstellen* wird für jeden Test der übersetzte Komponenten-Kode mit den zum Test gehörenden Test-Stubs und dem Test-Treiber zusammengebunden. Es entstehen lauffähige Binaries für jede Komponente.

Folgendes Produkt wird im Prozess *Ausführbare Komponente erstellen* modifiziert:

- Ausführbarer Komponenten-Kode.

Prozess *“Komponententest durchführen”*

Im Prozess *Komponententest durchführen* werden die lauffähigen Binaries ausgeführt. Während der Ausführung werden Testausgaben produziert und im Dokument E-Validation im Abschnitt “Dokumentation der Testergebnisse” beschrieben. Nach der Durchführung eines Tests werden die produzierten Testausgaben mit dem

erwarteten Verhalten der Komponente verglichen. Das erwartete Verhalten kann den Testplänen zum Komponententest entnommen werden. Gibt es Unterschiede zwischen dem tatsächlichem und dem erwarteten Verhalten einer Komponente wird dies als Fehlverhalten in der “Tabelle zur Bewertung der Testergebnisse” im Dokument E-Validation dokumentiert. Anschließend werden die beschriebenen Fehlverhalten auf Fehler in der System-Dokumentation zurückgeführt. Die Fehler werden in der Fehlertabelle im Dokument K-Verifikation festgehalten. Die Tabelle zur Bewertung der Testergebnisse wird um Verweise auf die Fehlernummern erweitert.

Folgendes Produkt wird im Prozess *Komponententest durchführen* konsumiert:

- Ausführbarer Komponente-Kode.

Folgende Produkte werden modifiziert:

- E-Validation.
- K-Verifikation.

Prozess *“Rework Komponententest”*

Im Prozess *Rework Komponententest* werden die in der System-Dokumentation lokalisierten und in der Fehlertabelle im Dokument K-Verifikation dokumentierten Fehler behoben. Die Strategien zur Behebung der Fehler werden in den Verifikationskapiteln der zu ändernden Dokumente beschrieben.

Folgende Produkte werden modifiziert:

- Komponenten-Kode.
- Ausführbare Komponente.
- K-Verifikation.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.

- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

5.7 Weiterführende Literatur

1. A. F. Ackerman, L. S. Buchwald, F. H. Lewski: *Software Inspections: An Effective Verification Process.* In: Donald J. Reifer (EDS.): Software Management. 5th Edition, IEEE Computer Society, Kapitel 10, S. 416-421, 1997.
2. Karsten Buch: *Richtlinien zur Ableitung von C++-Kode aus OMT-Statemodels der Entwurfsphase*. Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 67653 Kaiserslautern, Mai 1997.
3. Barbara Kitchenham, Steve Linkman: *Validation, Verification, and Testing: Diversity Rules*. IEEE Software, Vol. 15, Nr. 4, S. 46-49, Juli 1998.
4. Marco van Meegen, Pefer Schless (Hrsg.): *Programmkonventionen für C++*. Softwaretechnik-Trends, Band 12, Heft 3, S. 29-47, August 1992.

KAPITEL 6

System-Integration

Ziel des Prozesses *System-Integration* ist, aus den einzelnen ausführbaren Komponenten ein ausführbares Software-System zu erstellen. Dafür werden die Komponenten schrittweise bis zu einem ausführbaren System integriert. Entstehende Teilsysteme werden validiert.

Abbildung 96 gibt einen Überblick über die verwendeten Produkte im Prozess *System-Integration*.

Folgendes Produkt wird im Prozess *System-Integration* (in Teilen) produziert:

- *Ausführbares System.* Dieses Produkt umfasst die lauffähigen Teilsysteme (mit Test-Rahmen), das lauffähige Software-System (in der Entwicklungsumgebung) und das installierte Software-System (in der Anwendungsumgebung). Darüber hinaus beinhaltet das *ausführbare System* Makefiles, die das Zusammenbinden des übersetzten Komponenten-Kodes zu lauffähigen Produkten unterstützen.

Folgende Produkte werden modifiziert:

- *Problemlösungsdokumentation:* Dieses Dokument beinhaltet die Anforderungen an das zu entwickelnde Software-System aus Sicht der Benutzer/Kunden und der Entwickler.

- *Software-System-Dokumentation*. Dieses Dokument umfasst den UML-Entwurf für das zu implementierende Software-System. Außerdem beinhaltet es Anforderungen an die zu implementierenden Komponenten.
- *Software-Komponenten-Dokumentation:* Dieses Dokument umfasst den Quellkode der implementierten Komponenten.
- *Ausführbare Komponente*. Dieses Produkt beinhaltet den übersetzten Kode der Komponenten.

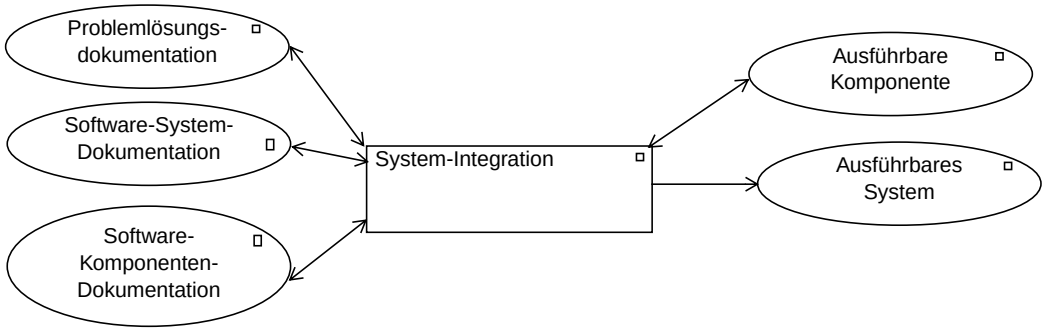


ABBILDUNG 96. Verwendete Produkte des Prozesses *System-Integration*

6.1 Prozessverfeinerung

Der Prozess *System-Integration* unterteilt sich in zwei Teilprozesse (siehe Abbildung 97). Ausgehend von dem ausführbaren Komponenten-Kode werden *Ausführbare Teilsysteme* erstellt. Der UML-Entwurf gibt Anleitung, welche Teilsysteme gebildet werden (im Entwurf als Packages gekennzeichnet). Außerdem kann dem UML-Entwurf entnommen werden, von welchen Komponenten die Teilsysteme abhängig sind, d. h. welche *Test-Rahmen* zum Ausführen der Teilsysteme benötigt werden. Die Funktionalität der *Test-Rahmen* wird anhand der Testpläne zum Test des Systems aus dem Dokument A-Validation bestimmt. In *Makefiles* werden die Komponenten festgehalten, die zum Zusammenbinden der *Ausführbaren Teilsysteme* benötigt werden. Die *Makefiles* erleichtern das Zusammenbinden der Komponenten zu *Ausführbaren Teilsystemen*. Im Prozess *System-Integration* werden die *Ausführbaren Teilsysteme* außerdem getestet. Entdeckte Fehlverhalten werden auf Fehler in der Dokumentation zurückgeführt und behoben.

Die *System-Integration* erfolgt iterativ (siehe Abbildung 98). Im ersten Iterationsschritt werden die Ausführbaren Komponenten zu *Ausführbaren Teilsystemen* integriert und die Teilsysteme werden getestet. In den darauffolgenden Iterationsschritten werden diese Teilsysteme zu umfassenderen *Ausführbaren Teilsystemen* integriert, die wiederum getestet werden. In einem letztem Iterationsschritt werden die verbleibenden Teilsysteme zu einem ausführbaren Gesamtsystem integriert. Letzteres wird nicht mehr im Rahmen des Prozesses *System-Integration* getestet, sondern in einem späteren Prozess (*System-Validation*). Mit den verschiedenen Iterationsschritten variieren die Produktflüsse in Abbildung 97. Daher ist an den Produktflussbeziehungen, die variieren können, jeweils gekennzeichnet, in welchen Iterationen sie gültig sind. Beispielsweise produziert der Teilprozess *Ausführbares Teilsystem erstellen* den *Ausführbaren Teilsystem-Kode* in der 1. bis zur (n-1). Iteration. In der n. Iteration (d. h., der letzten Iteration) wird dann der *Ausführbare System-Kode* produziert. Der Integrationstest wird in der n. Iteration nicht mehr durchgeführt.

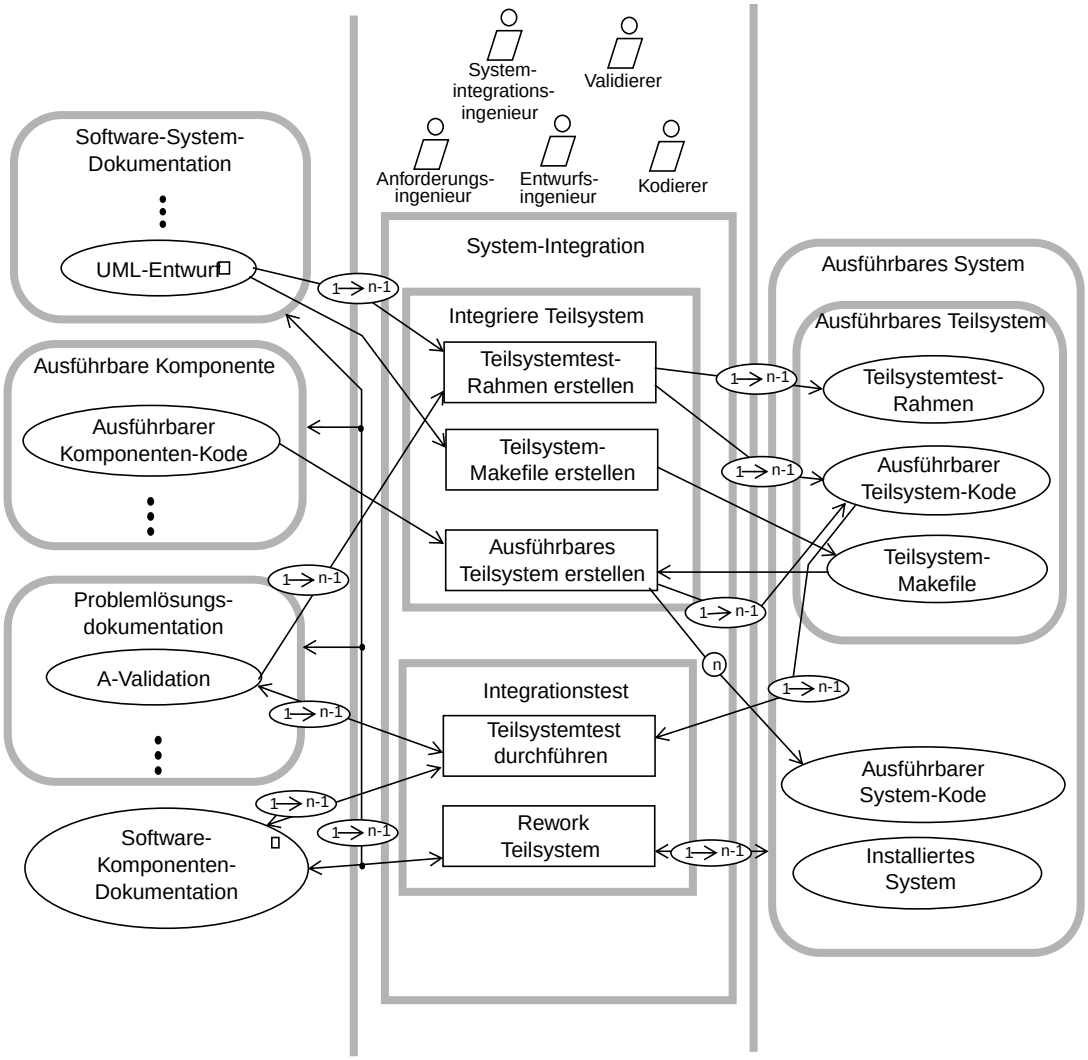


ABBILDUNG 97. Verfeinerung des Prozesses *System-Integration*

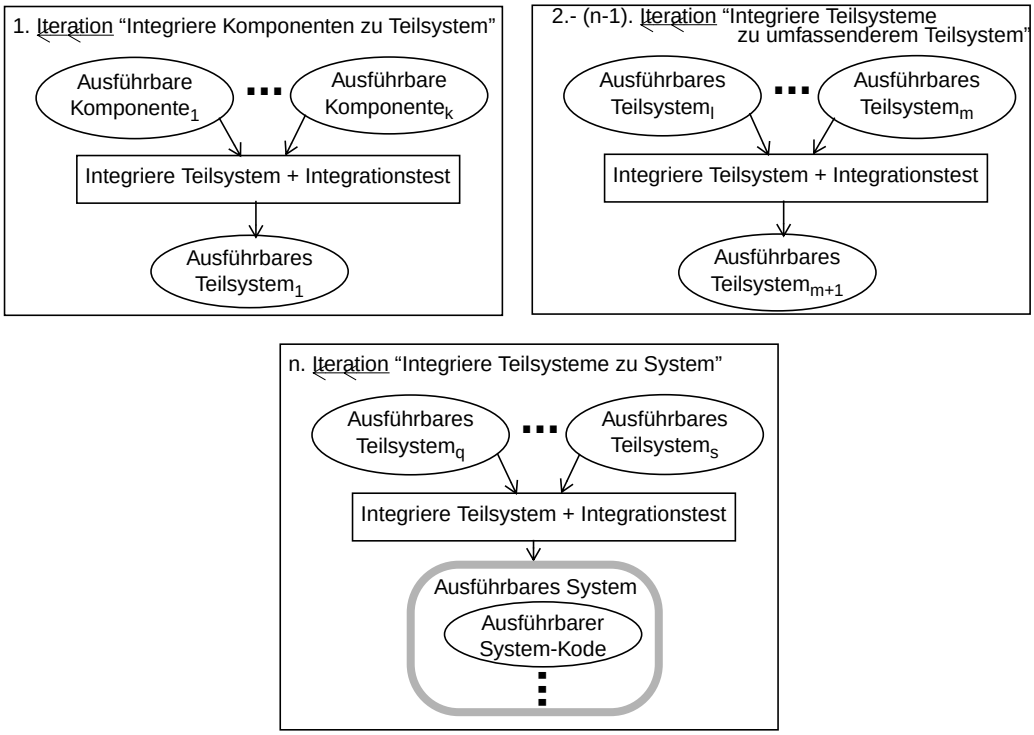


ABBILDUNG 98. Prinzip der *System-Integration*

6.2 Produkte

Im folgenden wird eine Übersicht über die im Prozess *System-Integration* verwendeten Produkte gegeben. Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte der Dokumente, soweit dies nicht bereits in vorangegangenen Kapiteln erfolgt ist.

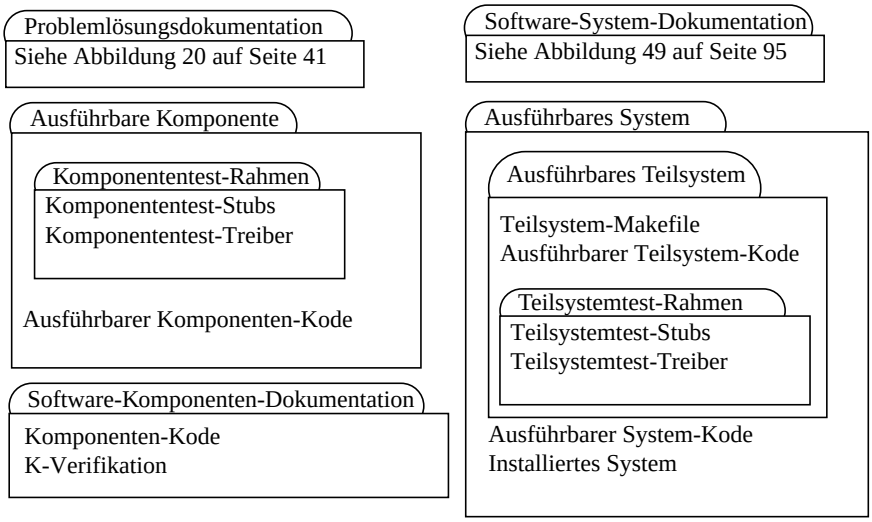


ABBILDUNG 99. Produktübersicht für den Prozess *System-Integration*

Ausführbares System

Das *Ausführbare System* umfasst alle zusammengesetzten ausführbaren Produkte, dies sind: die *Ausführbaren Teilsysteme*, der *Ausführbare Systemkode* und das *Installierte System*.

Wenn ein Software-System aus einer Vielzahl von Komponenten besteht, ist eine Integration aller Komponenten in einem Schritt meist nicht sinnvoll. Bei einer Integration in einem Schritt lassen sich beispielsweise Ursachen von Fehlverhalten des Systems nur sehr schwer lokalisieren. Aus diesem Grund sollte ein komplexes Software-System schrittweise integriert und getestet werden. Dafür werden *Ausführbare Teilsysteme* gebildet und deren Verhalten getestet. Teilsysteme sind nur ausführbar, wenn Abhängigkeiten zu Komponenten aus anderen Teilsystemen als *Treiber* und *Stubs* realisiert werden. Die Funktionalität der *Test-Rahmen* (d. h. der *Teilsystemtest-Stubs* und *-Treiber*) wird aus den Testplänen zum Test des Systems abgeleitet. Die *Teilsystem-Makefiles* unterstützen das Zusammenbinden des übersetzten Komponenten-Kodes (*Ausführbarer Komponenten-Kode*) zu ausführbaren zusammengesetzten Produkten.

Ausführbares Teilsystem

Das *Ausführbare Teilsystem* umfasst Produkte, die zur Ausführung von Teilsystemen notwendig sind. Dies sind: *Teilsystemtest-Rahmen*, *Teilsystem-Makefiles*, und der *ausführbare Teilsystem-Kode*.

Teilsystemtest-Rahmen

Der *Teilsystemtest-Rahmen* beinhaltet den Quellkode der Treiber und Stubs, die zum Test des Verhaltens der Teilsysteme benötigt werden. Die Abhängigkeiten eines Teilsystems nach außen sind im UML-Entwurf beschrieben. Diese Abhängigkeiten werden durch *Treiber* und *Stubs* realisiert. Die Funktionalität der *Test-Treiber* und *-Stubs* kann den Testplänen für den Systemtest (siehe Kapitel A-Validation) entnommen werden. Dies ist möglich, weil die Testpläne für den Systemtest aus den Teilsystem-spezifischen Szenarien abgeleitet wurden und damit das Verhalten einzelner Teilsysteme beschreiben.

Teilsystemtest-Treiber

Der *Teilsystemtest-Treiber* ist ein Modul (z. B. Teilsystemtest.cc) mit der Funktion main. Das Teilsystem wird initialisiert. Aufrufe des zu testenden Teilsystems durch andere Teilsysteme werden simuliert.

Abbildung 100 zeigt einen Ausschnitt aus einem *Teilsystemtest-Treiber* für das Teilsystem “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *Teilsystemtest-Treiber*

Dem UML-Entwurf kann entnommen werden, dass das Teilsystem “LichtregelungFlur” vom Teilsystem “Benutzungsschnittstelle” aufgerufen wird. Aufrufe durch die Benutzungsschnittstelle (d. h. durch die Klasse ControlPanelFm) werden durch den *Teilsystemtest-Treiber* simuliert. Darüber hinaus ist das Teilsystem aktiv, d. h. das Teilsystem fragt Sensorwerte laufend ab und reagiert auf Zustandsänderungen in Abhängigkeit bestimmter Zeitintervalle. Um dieses Verhalten zu beobachten, muss der Scheduler aktiviert und die Zeit initialisiert werden.

```
#include <Teilsystemtest.hh>
  int TestCase;
// Zeitpunkt (vergangene Minuten seit Simulationsstart), zu dem der Hausmeister ueber das ControlPanelFM das
Licht
// ein- (Testfall 6, 7) bzw. ausschalten (Testfall 8) soll.
int testCase6_pointOfTime_fMLightControl = 11;
int testCase7_ pointOfTime_fMLightControl = 11;
int testCase8_pointOfTime_fMLightControl = 11;
// Anzahl der Minuten, die ein bestimmter Testfall dauert
int timeDuration_testCase[9] = {0, 15, 30, 30, 0, 30, 20, 20, 20};
void main()
{
  House* HousePtr;
  Section* SectionPtr;
  cout << "Gruppe 3: Teilsystemtest\n";
  cout << "\n********** Initialisierung gestartet **********\n\n";
  cout << "Erzeugung des Schedulers...";
  Scheduler S(600000);
  cout << "erfolgreich beendet!\n";
  int StartHour = 0;
  int StartMinute = 0;
  Time StartTime(StartHour, StartMinute, 0, 1, 1, 1997);
  Timer SystemTime(StartTime, 60);
  Time CurrentTime;
  cout << "Erzeugung der Gebaeudestruktur...";
  HousePtr = init(SystemTime, S);
  cout << "erfolgreich beendet!\n";
  SectionPtr = HousePtr->GetSectionPtr(1);
  cout << "\n********** Initialisierung beendet **********\n\n\n";
  do {
        cout << "Welcher Testfall (1 - 3, 5 - 8) soll ausgefuehrt werden?" << endl;
        cin >> TestCase;
  } while ((TestCase < 1) || (TestCase > 8) || (TestCase == 4));
  cout << "\nTestfall " << TestCase << " wird ausgefuehrt!" << endl << endl;
  cout << "-------------------------------------------------" << endl;
  cout << "---------- Testlauf beginnt ----------" << endl;
  cout << "-------------------------------------------------" << endl << endl;
   // Variable, die festhaelt, ob der Aufruf HousePtr->enterFMLightControl(...) das erste Mal aufgerufen wird oder
nicht
  bool first_try = true;
  // Starten der Simulation durch Starten des Schedulers
  S.start();
  SystemTime.getActualTime(&CurrentTime);
 while ((CurrentTime.toDuration() / 60) <= timeDuration_TestCase[TestCase]){
   // Aktuelle Zeit ermitteln
   SystemTime.getActualTime(&CurrentTime);
   switch (TestCase){
   case 6:
     if (((Time.toDuration() / 60) >= testCase6_pointInTime_fMLightControl) && (first_try == true)) {
       // Der Hausmeister gibt ueber das ControlPanelFM das Kommando, das
       //Flurlicht fuer den Flurabschnitt 1 einzuschalten
       cout << "Der Hausmeister versucht ueber" << " das ControlPanelFm das Licht einzuschalten\n";
       HousePtr->enterFMLightControl(cLightOn, 1);
       // Der HousePtr->enterFmLightControl(...)-Aufruf soll nur einmal aufgerufen werden
       first_try = false;
     }
     break;
```

ABBILDUNG 100. Ausschnitt aus einem Teilsystemtest-Treiber

Teilsystemtest-Stub

Die *Test-Stubs* sind Dummies für Klassen, die durch das zu testende Teilsystem aufgerufen werden. Die Methoden, die die *Test-Stubs* zur Verfügung stellen müssen, können dem UML-Entwurf entnommen werden. Alle Methoden, die durch das zu testende Teilsystem aufgerufen werden, müssen durch *Stubs* simuliert werden. Die Funktionalität der *Teilsystemtest-Stubs*, d. h. Rückgabewerte und Ausgaben, werden durch die Testpläne zum Systemtest (siehe Kapitel A-Validation) bestimmt.

In einem Testplan wird ein bestimmter Startzustand vorausgesetzt. Dies kann z. B. bedeuten, dass die Anwesenheit einer Person im Raum gefordert wird. Initialisiert man ein Software-System, so stellt sich dieses auf einen Initialzustand ein, der fest kodiert ist (Initialzustände der State-Diagramme). Es ist wahrscheinlich, dass es Unterschiede zwischen dem geforderten Startzustand für den Test und dem Initialzustand des Software-Systems gibt. Für die *Stubs* heißt dies, dass es nötig sein kann, das Software-System zuerst in diesen Startzustand zu bringen. Erst dann folgen die Sensorwerte entsprechend des Testplans.

Abbildung 101 zeigt einen Ausschnitt aus einem *Teilsystemtest-Stub* für das Teilsystem “LichtRegelungFlur” aus dem Gebäudeautomationssystem.

Beispiel *Teilsystemtest-Stub*

Dem UML-Entwurf kann entnommen werden, dass das Teilsystem “LichtregelungFlur” nur Klassen aus dem Teilsystem “HardwareWrapper” aufruft. Die entsprechenden Klassen (d. h. Sensoren und Aktuatoren) werden durch *Teilsystemtest-Stubs* simuliert.

Als zentrale Anbindung an die konkrete Hardware wurde im UML-Entwurf des Gebäudeautomationssystems die Klasse Portbaustein eingeführt. Von dieser Klasse sind alle Sensor- und Aktuatorklassen abgeleitet. Beim Teilsystemtest dient diese Klasse zur Realisierung der *Stubs*. Die Methoden des Portbausteins werden so modifiziert, dass ein bestimmtes Verhalten der Sensoren erreicht wird.

Die Änderungen der Sensorwerte sind beim Teilsystemtest von der Zeit abhängig. Das Verhalten der Sensoren in Abhängigkeit der Zeit und des Testfalls wird im *Stub* durch die Definition eines Arrays mit zwei Dimensionen TESTFALL und

SCHRITTE eingeführt. Die erste Array-Dimension legt die Abhängigkeit der Sensorwerte von der Testfallnummer fest. In jedem Testfall haben die Sensoren ein unterschiedliches Verhalten. Die zweite Array-Dimension teil das Verhalten des Sensors in einzelne Schritte. Für jeden Schritt ist der Zeitpunkt der Wertänderung und der Sensorwert selbst anzugeben.

Wenn zu einem beliebigen Zeitpunkt ein Sensor abgefragt wird, kann der Rückgabewert mit Hilfe des Arrays bestimmt werden.

```
// * Testfall 1 (Flurlichtregelung durch Anwesenheit) *
// *****************************************************
// Wertefolge zum Erreichen des Startzustandes fuer den Testfall
// Keine Anwesenheit im Flurabschnitt
AnwesenheitskontaktImd1Door1[1][0].Minuten = 0;
AnwesenheitskontaktImd1Door1[1][0].Wert = offen;
AnwesenheitskontaktImd2Door1[1][0].Minuten = 0;
AnwesenheitskontaktImd2Door1[1][0].Wert = offen;
AnwesenheitskontaktImd1Door2[1][0].Minuten = 0;
AnwesenheitskontaktImd1Door2[1][0].Wert = offen;
AnwesenheitskontaktImd2Door2[1][0].Minuten = 0;
AnwesenheitskontaktImd2Door2[1][0].Wert = offen;
AnwesenheitskontaktImd3[1][0].Minuten = 0;
AnwesenheitskontaktImd3[1][0].Wert = offen;
// Kein Licht an im Flurabschnitt
LichtStatusFlur[1][0].Minuten = 0;
LichtStatusFlur[1][0].Wert = off;
// Startzustand erreicht
// Beginn des Tests
// Anwesenheit gemeldet durch AnwesenheitskontaktImd1Door1
AnwesenheitskontaktImd1Door1[1][1].Minuten = 11;
AnwesenheitskontaktImd1Door1[1][1].Wert = geschlossen;
// Ende des Tests bei 60 min
// Ende des Testfalls mit Werten belegen, damit Zeitdifferenz
// auch am Ende richtig berechnet werden kann.
AnwesenheitskontaktImd1Door1[1][2].Minuten = 60;
AnwesenheitskontaktImd1Door1[1][2].Wert = geschlossen;
AnwesenheitskontaktImd2Door1[1][1].Minuten = 60;
AnwesenheitskontaktImd2Door1[1][1].Wert = offen;
AnwesenheitskontaktImd1Door2[1][1].Minuten = 60;
AnwesenheitskontaktImd1Door2[1][1].Wert = offen;
AnwesenheitskontaktImd2Door2[1][1].Minuten = 60;
AnwesenheitskontaktImd2Door2[1][1].Wert = offen;
AnwesenheitskontaktImd3[1][1].Minuten = 60;
AnwesenheitskontaktImd3[1][1].Wert = offen;
// Kein Licht an im Flurabschnitt
LichtStatusFlur[1][1].Minuten = 60;
LichtStatusFlur[1][1].Wert = off;
```

ABBILDUNG 101. Ausschnitt aus einem Teilsystemtest-Stub

Teilsystem-Makefile

Die *Teilsystem-Makefiles* beschreiben Abhängigkeiten zwischen Komponenten innerhalb eines Teilsystems und vom Teilsystem zu anderen Komponenten. *Makefiles* erleichtern das Zusammenbinden von übersetzten Komponenten zu ausführbaren Teilsystemen. *Makefiles* können auch zum Kompilieren von Komponenten mit ihren Abhängigkeiten eingesetzt werden.

Abbildung 102 zeigt einen Ausschnitt aus einem *Teilsystem-Makefile* zum Zusammenbinden der Komponenten des Teilsystems “LichtRegelungFlur” des Gebäudeautomationssystems.

Beispiel *Teilsystem-Makefile*

Mit Hilfe des *Makefiles* zur Erstellung eines lauffähigen Teilsystems “LichtRegelungFlur” werden Treiber- und Stubs übersetzt. Anschließend werden die Komponenten zu einem ausführbaren Teilsystem (Teilsystemtest) zusammengebunden.

```
# Makefile
# C++-Compiler
CXX     = g++
# Praeprozessorflags
CPPFLAGS =  -I$(TESTDIR) -I$(TESTDIR)/../include -I$(TESTDIR)/../init
# C++-Compiler Flags
CXXFLAGS  =  -g
# Variablen fuer Dateien
# HIER BITTE DIE .cc UND DIE .o DATEIEN ERGAENZEN !!!!!
EXECUTABLES = Teilsystemtest
SRCS        = Teilsystemtest.cc Actuator.cc Contact.cc DimmableLight.cc OutdoorLight Sensor.cc Portbaustein.cc
RealRadiatorValve.cc RealBlind.cc RoomTemperatureSensor.cc Sensor.cc StatusLine.cc WaterTemperatureSensor.cc
init.cc
OBJS      = $(SRCS:%.cc=%.o)
GLOBALOBJS    = UserDataTempType.o UserDataDurationType.o Blind.o BlindController.o ClassDT2.o
ClassDT2Standby.o ClassTimeIntervalHeatingUpNormal.o ClassTimeIntervalHeatingUpStandbyNormal.o ClassTi-
meIntervalEntryStandbyNormal.o Clock.o ControlPanel.o ControlPanelFm.o Day.o DimmableLightController.o
HallwaySection.o House.o YearInterval.o Light.o LightController.o LightControlUnit.o PRBlindController.o Pres-
enceController.o PresenceControllerHallway.o PresenceControllerRoom.o PRHallwayLightController.o PRLight-
Controller.o PRTempController.o RadiatorController.o RadiatorValve.o Room.o RoomLightController.o
RoomLightControllerWithBlinds.o RoomLightControllerWithoutBlinds.o RoomLightRegulator.o RoomLightRegu-
latorWithBlinds.o RoomLightRegulatorWithoutBlinds.o RoomTemperatureController.o RoomWithBlinds.o Room-
WithoutBlinds.o ScheduledFct.o Scheduler.o Section.o Slist.o DayIntervals.o TemperatureData.o
Temperatureregulator.o TemperatureregulatorWithBlinds.o TemperatureregulatorWithoutBlinds.o DefinitionOfTem-
pOpModus.o Time.o Timer.o Holidays.o ValueInterface.o ValueInterfaceWithBlinds.o ValueInterfaceWithout-
Blinds.o Weekintervals.o TimeIntervalExit.o HeatingPeriod.o ClassTimeIntervalEntryNormal.o
# Linke die lokalen OBJS und die globalen GLOBALOBJS zum Systemtest
Teilsystemtest: $(OBJS)
    $(CXX) $(LDFLAGS) -o $@ $(OBJS) $(GLOBALOBJS_FULLPATH)
    @echo Teilsystemtest erfolgreich erzeugt
```

ABBILDUNG 102. Ausschnitt aus einem Teilsystem-Makefile

Ausführbarer Teilsystem-Kode

Der *Ausführbare Teilsystem-Kode* enthält für jedes Teilsystem die übersetzten Test-Rahmen. Außerdem umfasst er für jedes Teilsystem ein lauffähiges Binary zur Durchführung der Tests.

Ausführbarer System-Kode

Der *Ausführbare System-Kode* umfasst das lauffähige Binary für das Gesamtsystem.

Installiertes System

Das installierte System ist das beim Kunden eingesetzte System. Es interagiert mit den real existierenden Sensoren und Aktuatoren in der Systemumgebung.

6.3 Rollen

Im Prozess *System-Integration* sind im wesentlichen die Rollen *Systemintegrationsingenieur, Validierer, Anforderungsingenieur, Entwurfsingenieur, Kodierer* und *Qualitätsmanager* involviert. Die Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 103. Folgende wesentliche Tätigkeiten sind im Prozess *System-Integration* mit der Rollen verbunden:

- ***Systemintegrationsingenieur.*** Der Systemintegrationsingenieur integriert iterativ übersetzten Komponenten-Kode zu *Ausführbaren Teilsystemen* und schließlich zum Gesamtsystem.
- ***Validierer.*** Die Aufgaben des Validierers im Prozess *System-Integration* sind, *Teilsystemtest-Rahmen* zu erstellen, den Teilsystemtest durchzuführen und die Ergebnisse zu dokumentieren.
- ***Kodierer.*** Die Aufgabe des Kodierers ist, beim Teilsystemtest gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Anforderungsingenieur.*** Die Aufgabe des Anforderungsingenieurs ist, beim Teilsystemtest gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben. Hierfür kann er den Kunden kontaktieren, sofern die *Problembeschreibung* oder die *Informellen B-Anforderungen* betroffen sind.
- ***Entwurfsingenieur.*** Die Aufgabe des Entwurfsingenieurs ist, beim Teilsystemtest gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Qualitätsmanager.*** Die Aufgabe des Qualitätsmanagers ist, entsprechend den längerfristigen Bedürfnissen und Erwartungen einer Software-Entwicklungsorganisation Wissen/Erfahrung in geeigneter Form zu verwalten, einzelnen Projekten auf Anfrage zur Wiederverwendung zur Verfügung zu stellen und kontinuierlich aufgrund gemachter Projekterfahrungen zu verbessern. Wissen/Erfahrungen kann vom Qualitätsmanager im Prozess *System-Integration* z. B. in Form von Fehlerklassifikationen zur Verfügung gestellt werden. Diese sind zum Teil Bestandteil von Produkten, für die andere Rollen verantwortlich sind.

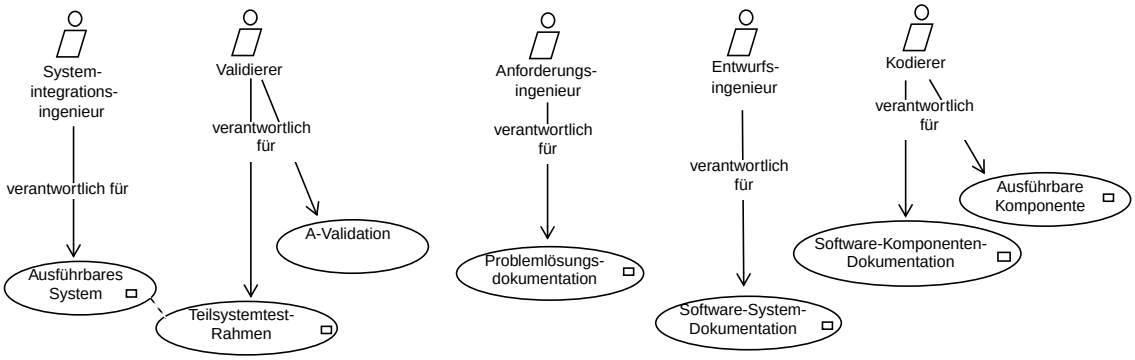


ABBILDUNG 103. Ressourcen und Produkte des Prozesses *System-Integration*

6.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 104 dargestellt. Die Iterationen, in denen die Kontrollflüsse gültig sind, sind wiederum an den Pfeilen angegeben.

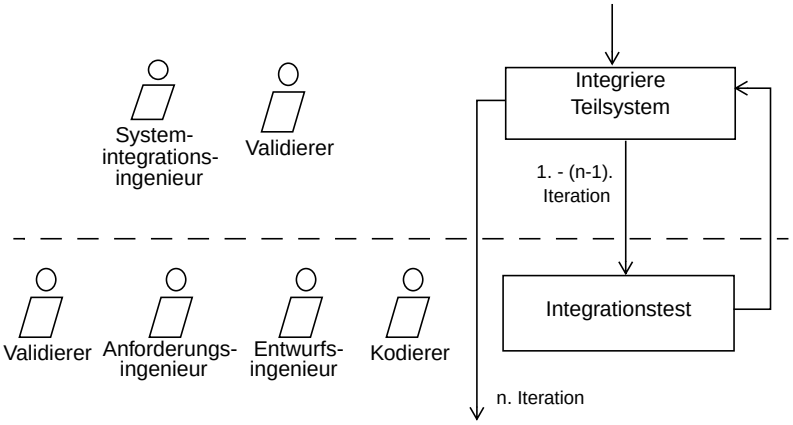


ABBILDUNG 104. Kontrollfluss für den Prozess *System-Integration*

6.5 Prozess “Integriere Teilsystem”

Im Prozess *Integriere Teilsystem* werden aus dem übersetzten Komponenten-Kode Ausführbare Teilsysteme konstruiert. Im UML-Entwurf ist beschrieben, welche Teilsysteme gebildet werden. Basierend auf dem UML-Entwurf und den Testplänen zum Systemtest werden Test-Rahmen für die Teilsysteme entwickelt. Die Definition von Makefiles erleichtert ein Zusammenbinden der Komponenten zu Ausführbaren Teilsystemen.

Abbildung 105 zeigt den Prozess *Integriere Teilsystem* mit konsumierenden, produzierenden und modifizierenden Produkten.

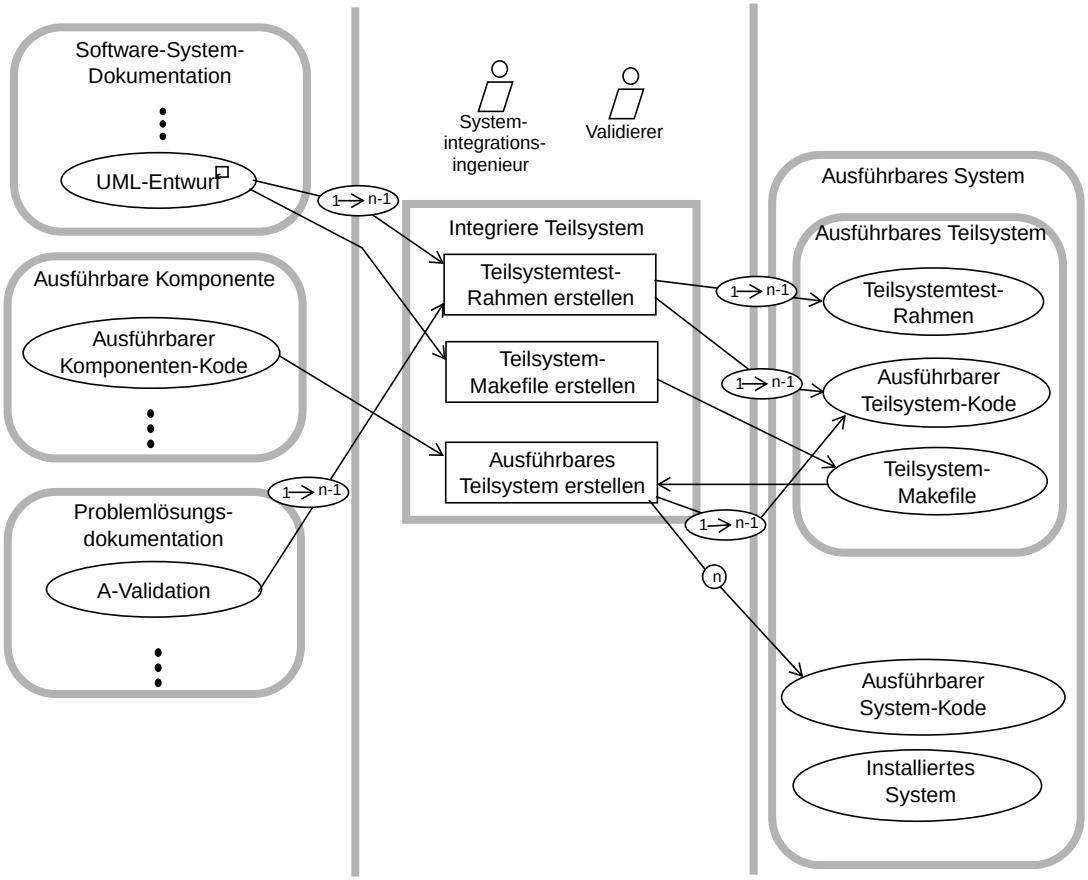


ABBILDUNG 105. Verwendete Produkte im Prozess *Integriere Teilsystem*

Folgende Produkte werden im Prozess *Integriere Teilsystem* konsumiert:

- Ausführbarer Komponenten-Kode.
- UML-Entwurf.
- A-Validation.

Folgende Produkte werden produziert:

- Teilsystemtest-Rahmen.

- Ausführbarer Teilsystem-Kode (1.-(n-1). Iteration).
- Ausführbarer System-Kode (n. Iteration).

Folgende Produkte werden modifiziert:

- Teilsystem-Makefile.

Prozess *“Teilsystemtest-Rahmen erstellen”*

Im Prozess *Teilsystemtest-Rahmen erstellen* werden zum Test der Teilsysteme Test-Treiber und -Stubs implementiert und übersetzt. Der UML-Entwurf zeigt, welche Treiber (d. h. Komponenten, die das Teilsystem aufrufen) und Stubs (d. h. Komponenten, die vom Teilsystem aufgerufen werden) implementiert werden müssen. Die Testpläne für den Systemtest (siehe Dokument A-Validation) legen fest, welche Funktionalität die Test-Treiber und Test-Stubs bieten müssen.

Folgende Produkte werden im Prozess *Teilsystemtest-Rahmen erstellen* konsumiert:

- UML-Entwurf.
- A-Validation.

Folgende Produkte werden produziert:

- Teilsystemtest-Rahmen. Hier wird der Quellkode der Treiber und Stubs dokumentiert.
- Ausführbarer Teilsystem-Kode. In diesem Produkt befinden sich die übersetzten Treiber und Stubs.

Prozess *“Teilsystem-Makefile erstellen”*

Im Prozess *Teilsystem-Makefile erstellen* werden Makefiles zum Zusammenbinden von Komponenten eines Teilsystems mit dem passenden Treiber und den passenden Stubs zu einem ausführbaren Teilsystem erstellt.

Folgendes Produkt wird konsumiert:

- UML-Entwurf.

Folgendes Produkt wird produziert:

- Teilsystem-Makefile.

Prozess *“Ausführbares Teilsystem erstellen”*

Im Prozess *Ausführbares Teilsystem erstellen* wird das Makefile verwendet, um ausführbare Teilsysteme zu erstellen. Der übersetzte Kode der Komponente wird mit dem übersetzten Treiber und den Stubs zusammengebunden. Es entstehen lauffähige Binaries für jedes Teilsystem.

Folgende Produkte werden konsumiert:

- Ausführbarer Komponenten-Kode
- Teilsystem-Makefile.

Folgende Produkte werden produziert:

- Ausführbarer Teilsystem-Kode (1.-(n-1). Iteration). Hierbei handelt es sich um das neue, umfassendere Teilsystem.
- Ausführbarer System-Kode (n. Iteration). Hierbei handelt es sich um das Gesamtsystem.

6.6 Prozess “Integrationstest”

Im Prozess *Integrationstest* werden die lauffähigen Binaries ausgeführt und Testergebnisse werden protokolliert, auf Fehler in der Dokumentation zurückgeführt und behoben. Dieser Prozess wird nicht in der letzten Iteration (d. h. nach der Integration zum Gesamtsystem) durchgeführt.

Abbildung 106 zeigt den Prozess *Integrationstest* mit konsumierenden, produzierenden und modifizierenden Produkten.

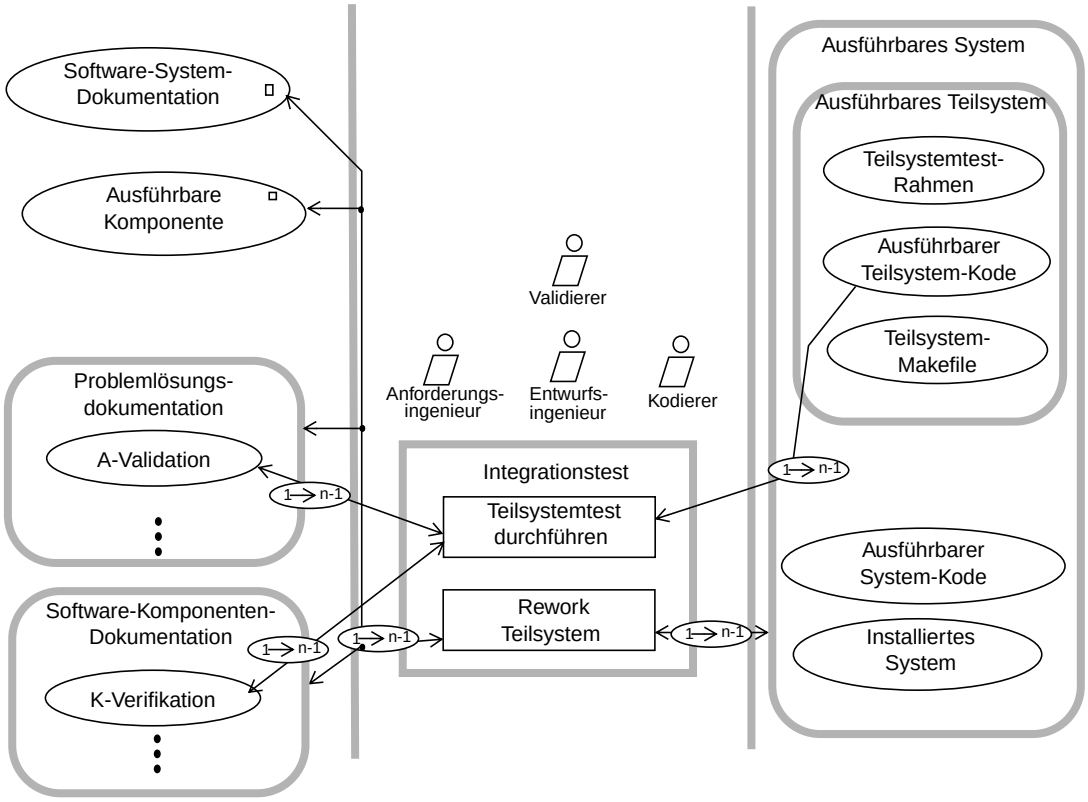


ABBILDUNG 106. Verwendete Produkte im Prozess *Integrationstest*

Folgende Produkte werden im Prozess *Integrationstest* modifiziert:

- Problemlösungsdokumentation.
- Software-System-Dokumentation.
- Software-Komponenten-Dokumentation.
- Ausführbare Komponente.
- Ausführbares System.

Prozess *"Teilsystemtest durchführen"*

Im Prozess *Teilsystemtest durchführen* werden die lauffähigen Teilsystem-Binaries ausgeführt. Während der Ausführung werden Testausgaben produziert und im Dokument A-Validation im Abschnitt "Dokumentation der Testergebnisse" beschrieben. Nach der Durchführung eines Tests werden die produzierten Testausgaben mit dem erwarteten Verhalten des Teilsystems verglichen. Das erwartete Verhalten kann den Testplänen zum Systemtest entnommen werden. Gibt es Unterschiede zwischen dem tatsächlichem und dem erwarteten Verhalten eines Teilsystems, wird dies als Fehlverhalten in der "Tabelle zur Bewertung der Testergebnisse" im Dokument A-Validation dokumentiert. Anschließend werden die beschriebenen Fehlverhalten auf Fehler in der System-Dokumentation zurückgeführt. Die Fehler werden in der Fehlertabelle im Dokument K-Verifikation festgehalten. Die Tabelle zur Bewertung der Testergebnisse wird um Verweise auf die Fehlernummern erweitert.

Folgende Produkte werden im Prozess "Teilsystemtest durchführen" konsumiert:

- Ausführbarer Teilsystem-Kode.

Folgende Produkte werden modifiziert:

- A-Validation.
- K-Verifikation.

Prozess *"Rework Teilsystem"*

Im Prozess *Rework Teilsystem* werden die in der System-Dokumentation lokalisierten und in der Fehlertabelle im Dokument K-Verifikation dokumentierten Fehler behoben. Die Strategie zur Behebung der Fehler wird in den Verifikationskapiteln der zu ändernden Dokumente beschrieben.

Folgende Produkte werden modifiziert:

- Komponenten-Kode.
- K-Verifikation.
- Ausführbare Komponente.
- Ausführbares Teilsystem.
- UML-Entwurf.
- Komponenten-Anforderungen.

- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

6.7 Weiterführende Literatur

1. Peter Liggesmeyer: *Modultest und Modulverifikation: state of the art*. BI-Wiss.-Verlag, Mannheim, Wien, Zürich 1990.
2. Glenford J. Meyers: *Methodisches Testen von Programmen.* Oldenbourg Verlag, München, Wien 1982.

KAPITEL 7

System-Validation

Ziel des Prozesses *System-Validation* ist, das *ausführbare System* in der Entwicklungs- und anschließend in der Anwendungsumgebung gegen die Anforderungen zu testen. Der Test in der Entwicklungsumgebung wird als *Systemtest* bezeichnet. Dabei wird die Zuverlässigkeit des Systems hinsichtlich funktionaler (Funktionstest) und nicht-funktionaler Anforderungen, wie z. B. Performanz überprüft. Der Test in der Anwendungsumgebung wird *Akzeptanz-* oder *Abnahmetest* genannt. Besteht das System diesen Test, nimmt der Kunde das System ab und es wird beim Kunden eingeführt. Treten während der Tests Fehlverhalten des Systems auf, werden diese auf Fehler in der Dokumentation zurückgeführt und behoben.

Abbildung 107 gibt einen Überblick über die verwendeten Produkte im Prozess *System-Validation*.

Folgende Produkte werden im Prozess *System-Validation* konsumiert:

- *Simulator.* Der *Simulator* ist ein Software-System zur Simulation der Systemumgebung. Das Verhalten der Aktuatoren und Sensoren wird simuliert. Die Schnittstelle des *Simulators* zum entwickelten Software-System sollte der Schnittstelle der realen Aktuatoren und Sensoren entsprechen.

- *Reale Systemumgebung.* Die *reale Systemumgebung* umfasst die real-existierenden Sensoren und Aktuatoren. In der Umgebung gelten die Beschränkungen der realen Welt. Die *reale Systemumgebung* ist nicht tatsächlich ein Produkt. Sie wurde an dieser Stelle als Produkt modelliert, um die verschiedenen Tests des Systems zu beschreiben.

Folgende Produkte werden modifiziert:

- *Problemlösungsdokumentation*. Dieses Dokument beinhaltet die Anforderungen an das zu entwickelnde Software-System aus Sicht der Benutzer/Kunden und der Entwickler.
- *Software-System-Dokumentation.* Dieses Dokument umfasst den UML-Entwurf für das zu implementierende Software-System. Außerdem beinhaltet es Anforderungen an die zu implementierenden Komponenten im System.
- *Software-Komponenten-Dokumentation*. Dieses Dokument umfasst den Quellkode der implementierten Komponenten.
- *Ausführbare Komponente*. Dieses Produkt umfasst den übersetzten und mit einem Treiber und einer Menge Stubs zusammengebundenen Kode der Komponenten.
- *Ausführbares System*. Dieses Produkt umfasst alle zusammengesetzten ausführbaren Produkte. Dies sind die ausführbaren Teilsysteme, das ausführbare System und das installierte System.

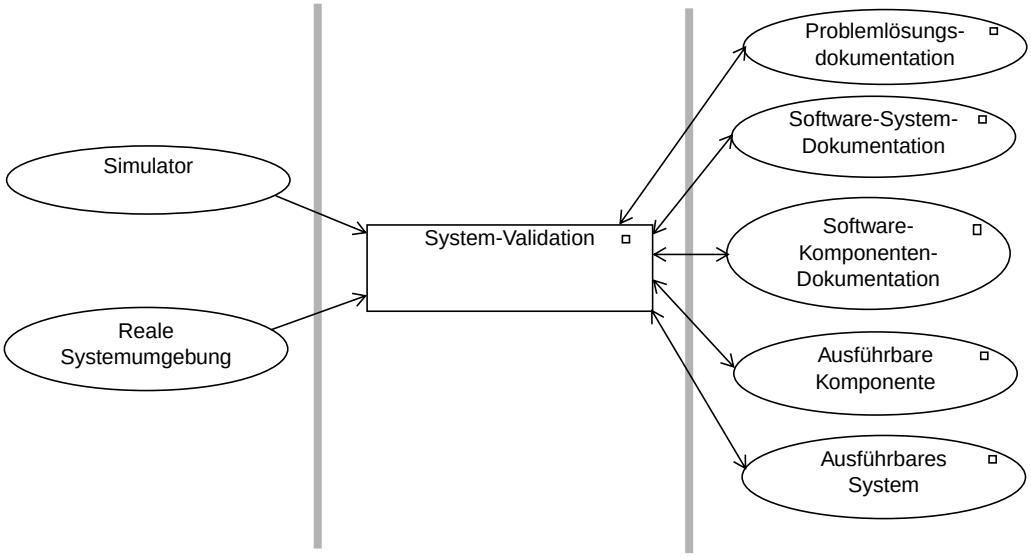


ABBILDUNG 107. Verwendete Produkte des Prozesses *System-Validation*

7.1 Prozessverfeinerung

Der Prozess *System-Validation* unterteilt sich in drei Teilprozesse (siehe Abbildung 108). Im Prozess *Systemtest* wird das System gegen die B-Anforderungen unter Verwendung des *Simulators* getestet. Der *Simulator* simuliert dabei die Systemumgebung. Während des Prozesses *Akzeptanztest* wird das System gegen die B-Anforderungen in der Anwendungsumgebung (d. h. der *realen Systemumgebung*) getestet. Real existierende Sensoren werden gelesen und Aktuatoren angesteuert. Wenn der Kunde das System nach dem Test akzeptiert, wird es in den laufenden Betrieb übernommen. Es folgt der Prozess *Test (des Systems) im Betrieb.* Wenn während der verschiedenen Tests Fehlverhalten des Systems auftreten, werden diese auf Fehler in der Dokumentation zurückgeführt und behoben.

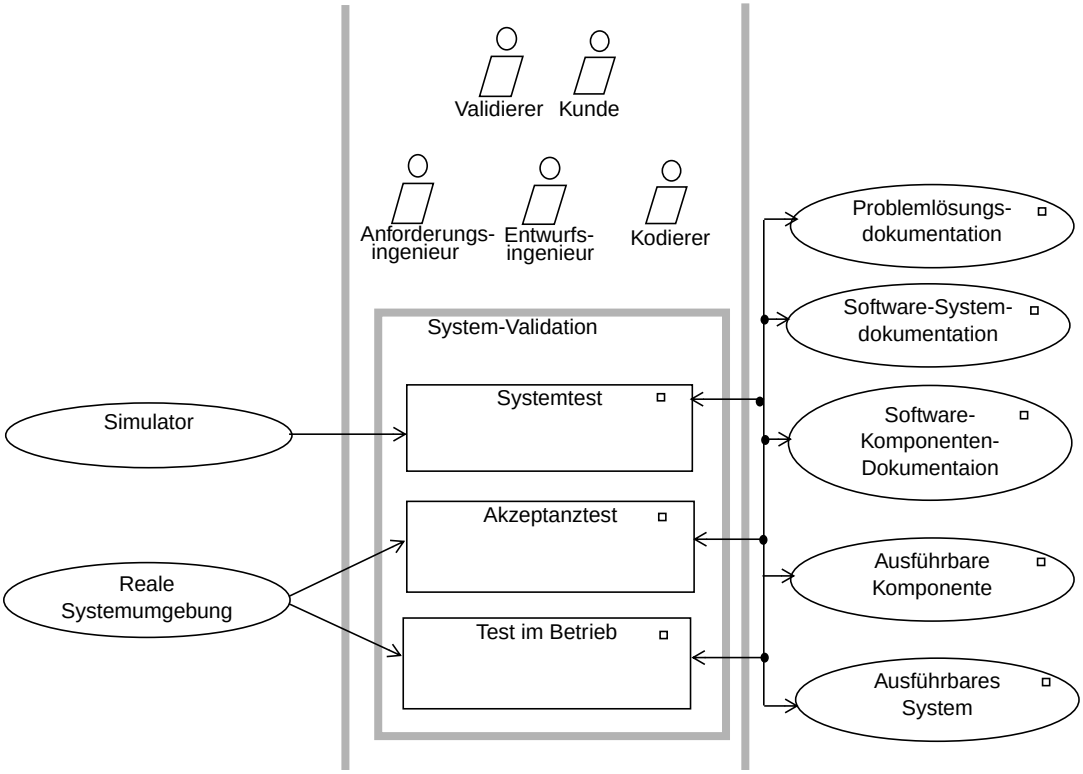


ABBILDUNG 108. Verfeinerung des Prozesses *System-Validation*

7.2 Produkte

Im folgenden wird eine Übersicht über die im Prozess *System-Validation* verwendeten Produkte gegeben. Die Übersicht beschreibt die interne Strukturierung der Dokumente. Anschließend erfolgt eine Beschreibung der Inhalte der Dokumente, soweit dies nicht bereits in vorangegangenen Kapiteln erfolgt ist.

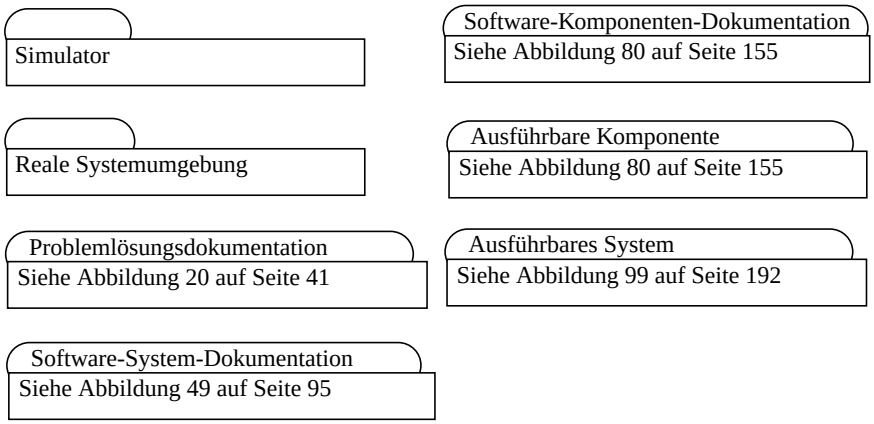


ABBILDUNG 109. Produktübersicht für den Prozess *System-Validation*

Simulator

Der Test eines Systems in der realen Systemumgebung ist oft schwierig. Zum einen kann eine falsche Ansteuerung der Aktuatoren zu Schäden in der Systemumgebung führen, z. B. am Gebäude und am Inventar eines Raumes. Zum anderen müssen die herrschenden Umweltbedingungen in der realen Systemumgebung akzeptiert werden. Umweltbedingungen lassen sich in der Regel nicht vollständig beeinflussen. Dies erschwert eine Überprüfung des Systems gegen die Anforderungen. Ein bestimmtes Verhalten des Systems kann nur getestet werden, wenn bestimmte Umweltbedingungen gelten (beschrieben als Vorbedingung zur Ausführung eines Szenario). Das Verhalten lässt sich in der realen Umgebung also nur dann prüfen, wenn die realen Umweltbedingungen zufällig mit den im Test geforderten Umweltbedingungen übereinstimmen.

Der Einsatz eines *Simulators*, der das Verhalten der real-existierenden Sensoren und Aktuatoren und der Systemumgebung simuliert, beseitigt die Probleme. Eine falsche Ansteuerung von Aktuatoren bleibt ohne Folge und bestimmte Umweltbedingungen lassen sich künstlich herstellen. Außerdem lassen sich Reaktionszeiten in der Umgebung (z. B. das Aufheizen eines Raums) künstlich beschleunigen.

Beim Einsatz eines *Simulators* muss gewährleistet sein, dass die Schnittstelle, die dem entwickelten Software-Systems angeboten wird, mit der Schnittstelle der real-existierenden Sensoren und Aktuatoren übereinstimmt.

Abbildung 110 zeigt einen Ausschnitt aus der Beschreibung des im Gebäudeautomationsprojekt verwendeten *Simulators*.

Beispiel *Simulator*

Der Simulator simuliert die Sensoren und Aktuatoren im Gebäude. Er stellt eine graphische Oberfläche zum Anzeigen und Manipulieren der Aktuatoren und Sensoren zur Verfügung.

Der Gebäude-Simulator ist ein ERLANG-Programm. Es bietet folgendes:

- Basisfunktionalität für eine diskrete Simulation.
- Funktionaltität für eine diskrete Simulation von kontinuierlichen Systemen.
- Eine Menge von Simulationsobjekten, die nützlich sind, um physikalische Effekte in Gebäuden zu simulieren.
- Verschiedene Hilfsmittel um auf den Zustand der Simulationsobjekte während der Simulation zuzugreifen.

Die Schnittstelle des Simulators entspricht der Schnittstelle der tatsächlichen Aktuatoren und Sensoren.

Im “BuildungSimulationWindow” können verschiedene Objekte (z. B. Fenster, Heizkörper, Räume und die Umgebung des Hauses) angewählt und über den Button “Objekt Fenster” geöffnet werden. Daraufhin werden die Eigenschaften dieser Objekte in eigenen Fenstern angezeigt. In diesen Fenstern kann man die Zustände der einzelnen Aktuatoren ablesen und die Werte der Sensoren verändern.

ABBILDUNG 110. Ausschnitt aus der Beschreibung eines Simulators

Reale Systemumgebung

Die *reale Systemumgebung* umfasst die real-existierenden Sensoren und Aktuatoren im Gebäude. In der *realen Systemumgebung* herrschen die Beschränkungen der realen Welt, die nicht beeinflusst werden können (z. B. das Aufheizen des Raumes dauert eine Zeitspanne t, die sich nicht verkürzen lässt). Diese Beschränkungen müssen beim Test in der *realen Umgebung* akzeptiert werden und führen dazu, dass bestimmte Systemverhalten nicht getestet werden können, weil sich bestimmte Umweltbedingungen nicht herstellen lassen (z. B. lässt sich im Sommer die Systemreaktion auf eine Raumtemperatur unter 4 Grad nur selten testen).

7.3 Rollen

Im Prozess *System-Validation* sind im wesentlichen die Rollen *Validierer, Kunde, Anforderungsingenieur, Entwurfsingenieur, Kodierer* und *Systemintegrationsingenieur* involviert. Die Verantwortlichkeiten für die Produkte ergeben sich aus Abbildung 111. Folgende wesentliche Tätigkeiten sind im Prozess *System-Validation* mit den Rollen verbunden:

- ***Validierer.*** Die Aufgabe des Validierers ist es, den *System-* und *Akzeptanztest* durchzuführen und die Ergebnisse zu dokumentieren. Beim *Test im Betrieb* sind die vom Kunden entdeckten Fehlverhalten zu dokumentieren. Des Weiteren sind jeweils Maßnahmen zur Fehlerbehebung einzuleiten. Darüber hinaus ist der Validierer verantwortlich für die Bereitstellung eines Simulators.
- ***Kunde.*** Die Aufgabe des Kunden beim *Akzeptanztest* ist, das System abzunehmen. Beim *Test im Betrieb* ist es Aufgabe des Kunden, im Betrieb auftretende Fehlermeldungen der Software-Entwicklungsorganisation zu melden.
- ***Anforderungsingenieur.*** Die Aufgabe des Anforderungsingenieurs ist, bei den Tests gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Entwurfsingenieur.*** Die Aufgabe des Entwurfsingenieurs ist, bei den Tests gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Kodierer.*** Die Aufgabe des Kodierers ist, bei den Tests gefundene Fehler in den ihm zugeordneten Dokumenten zu beheben.
- ***Systemintegrationsingenieur.*** Die Aufgabe des Systemintegrationsingenieurs ist, bei den Tests gefundene Fehler im *Ausführbaren System* zu beheben.
- ***Qualitätsmanager.*** Die Aufgabe des Qualitätsmanagers ist, entsprechend den längerfristigen Bedürfnissen und Erwartungen einer Software-Entwicklungsorganisation Wissen/Erfahrung in geeigneter Form zu verwalten, einzelnen Projekten auf Anfrage zur Wiederverwendung zur Verfügung zu stellen und kontinuierlich aufgrund gemachter Projekterfahrungen zu verbessern.

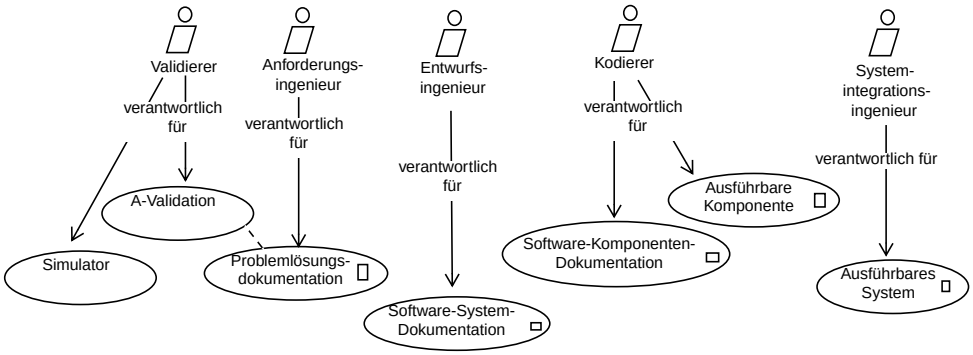


ABBILDUNG 111. Ressourcen und Produkte des Prozesses *System-Validation*

7.4 Kontrollfluss

Die Reihenfolge der Bearbeitung der Teilprozesse ist in Form eines Präzedenzgraphen in Abbildung 112 dargestellt.

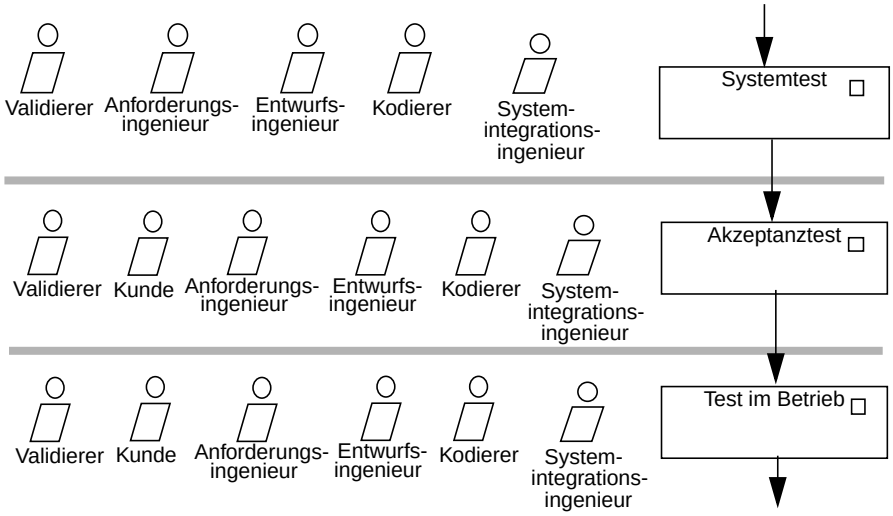


ABBILDUNG 112. Kontrollfluss für den Prozess *System-Validation*

7.5 Prozess “Systemtest”

Im Prozess *Systemtest* wird der ausführbare Systemkode in der Entwicklungsumgebung getestet. Die Systemumgebung wird durch einen Simulator simuliert. Die Zuverlässigkeit des Systems wird hinsichtlich der funktionalen (Funktionstest) und nicht-funktionalen Anforderungen überprüft. Als Grundlage für den Test dienen die Testpläne im Dokument A-Validation. Das Verhalten des Systems bei der Testdurchführung wird protokolliert und entdeckte Fehlverhalten werden auf Fehler in der Dokumentation zurückgeführt und behoben.

Abbildung 113 zeigt den Prozess *Systemtest* mit konsumierenden, produzierenden und modifizierenden Produkten.

Folgendes Produkt wird im Prozess *Systemtest* konsumiert:

- Simulator.

Folgende Produkte werden modifiziert:

- Problemlösungsdokumentation.
- Software-Systemdokumentation.
- Software-Komponenten-Dokumentation.
- Ausführbare Komponente.
- Ausführbares System.

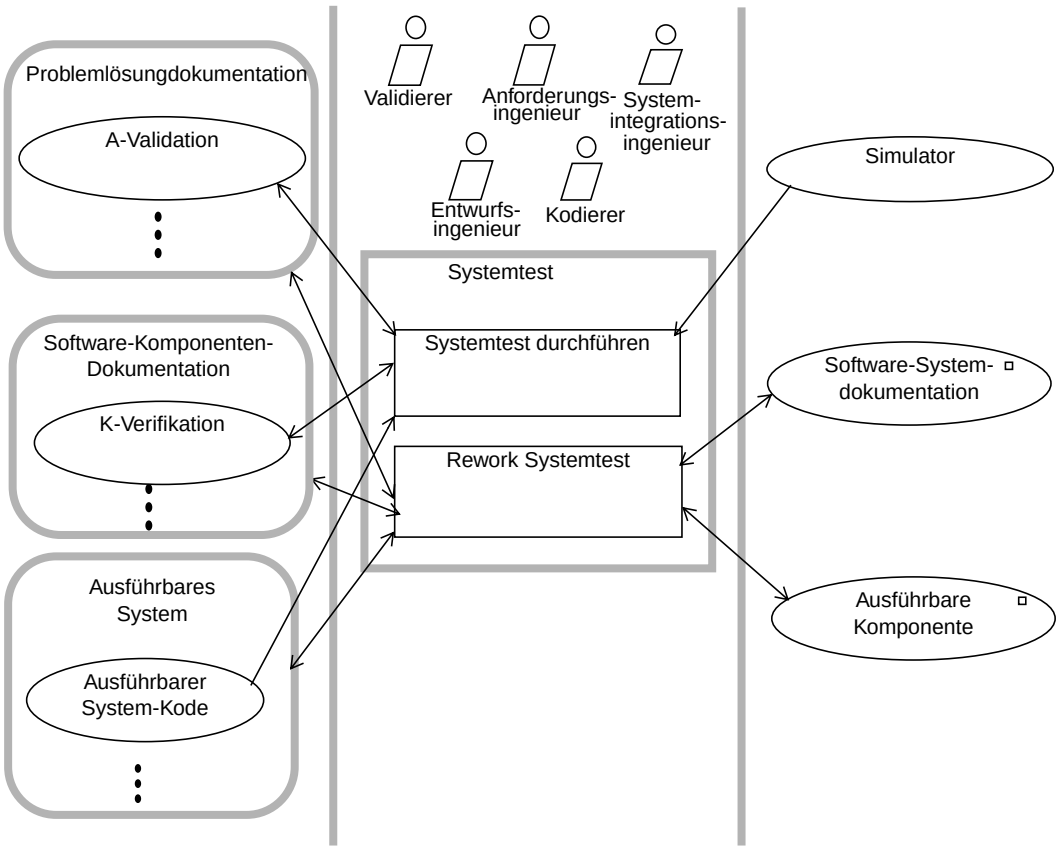


ABBILDUNG 113. Verwendete Produkte des Prozesses *Systemtest*

Prozess *“Systemtest durchführen”*

Im Prozess *Systemtest durchführen* wird das System unter Verwendung des Simulators ausgeführt. Durch den Einsatz des Simulators lassen sich die in den Testplänen beschriebenen Vorbedingungen künstlich herstellen. Durch die Manipulation der Sensorwerte im Simulator werden die in den Testplänen beschriebenen Ausgangsbedingungen geschaffen. Das Verhalten des Systems wird beobachtet und im Dokument A-Validation im Abschnitt “Dokumentation der Testergebnisse” protokolliert. Nach der Durchführung eines Tests wird das protokollierte Systemverhalten mit dem erwarteten Verhalten des Systems verglichen. Das erwartete Verhalten kann den Testplänen entnommen werden. Gibt es Unterschiede zwischen dem tatsächlichen und dem erwarteten Verhalten des Systems wird dies als Fehlverhalten in der “Tabelle zur Bewertung der Testergebnisse” im Dokument A-Validation dokumentiert. Anschließend werden die beschriebenen Fehlverhalten auf Fehler in der System-Dokumentation zurückgeführt und die Fehler werden in der “Fehlertabelle” im Dokument K-Verifikation festgehalten. Die Tabelle zur Bewertung der Testergebnisse wird um Verweise auf die Fehlernummern erweitert.

Folgende Produkte werden im Prozess *Systemtest durchführen* konsumiert:

- Simulator.
- Ausführbarer System-Kode.

Folgende Produkte werden modifiziert:

- A-Validation.
- K-Verifikation.

Prozess *“Rework Systemtest”*

Im Prozess *Rework Systemtest* werden die in der System-Dokumentation lokalisierten und in der Fehlertabelle im Dokument K-Verifikation dokumentierten Fehler behoben. Die Strategie zur Behebung der Fehler wird in den Verifikationskapiteln der zu ändernden Dokumente beschrieben.

Folgende Produkte werden im Prozess Rework Systemtest modifiziert:

- Komponenten-Kode.
- K-Verifikation.
- Ausführbare Komponente.
- Ausführbares System.

- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

7.6 Prozess "Akzeptanztest"

Im Prozess *Akzeptanztest* wird das System in der Anwendungsumgebung, d. h. der realen Systemumgebung beim Kunden, getestet. Dafür muss das System zuerst beim Kunden installiert werden (Prozess *Installiere System*). Das Ergebnis des Akzeptanztests entscheidet über die Abnahme des Systems durch den Kunden. Im Gegensatz zur Verwendung des Simulators lassen sich die realen Umgebungsbedingungen beim Test nicht beeinflussen. Entdeckte Fehlverhalten werden protokolliert, auf Fehler in der Dokumentation zurückgeführt und behoben.

Abbildung 113 zeigt den Prozess *Akzeptanztest* mit konsumierenden, produzierenden und modifizierenden Produkten.

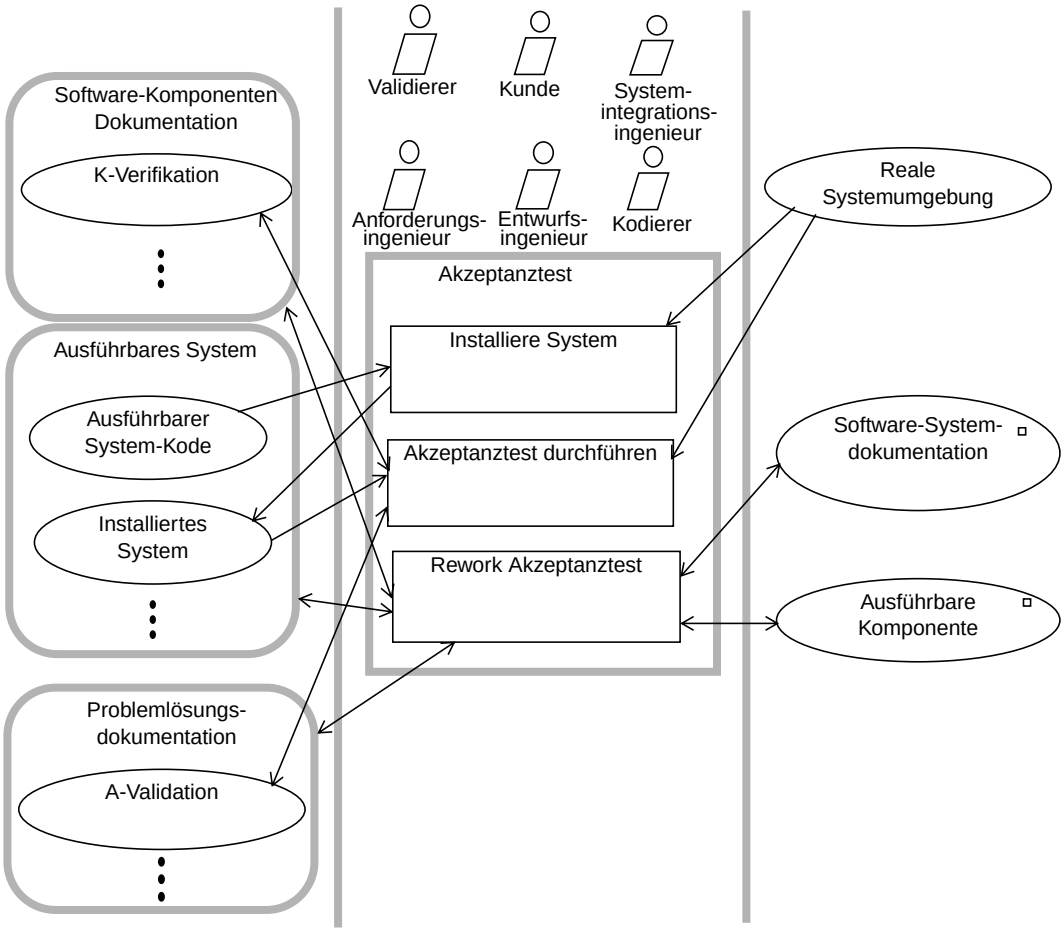


ABBILDUNG 114. Verwendete Produkte des Prozesses *Akzeptanztest*

Prozess *“Installiere System”*

Im Prozess *Installiere System* wird das System beim Kunden eingerichtet, d. h. Sensoren und Aktuatoren werden installiert und das entwickelte Software-System wird auf den Zielrechner portiert.

Folgendes Produkte werden im Prozess *Installiere System* konsumiert:

- Ausführbarer System-Kode.
- Reale Systemumgebung.

Folgendes Produkt wird produziert:

- Installiertes System.

Prozess *“Akzeptanztest durchführen”*

Im Prozess *Akzeptanztest durchführen* wird das System in der realen Systemumgebung getestet. Gewöhnlich wird dieser Test durch den Kunden/Benutzer durchgeführt. Dabei wird das Verhalten des installierten Systems mit dem in den B-Anforderungen beschriebenen Verhalten verglichen. Das installierte System wird im Einsatz getestet. Wenn das System den Test besteht, dann wird das System vom Kunden akzeptiert. Zur Durchführung des Tests kann der Kunde die Testpläne aus dem Dokument A-Validation verwenden.

Das während des Akzeptanztests gezeigte Verhalten des Systems wird im Dokument A-Validation im Abschnitt “Dokumentation der Testergebnisse” beschrieben und mit dem erwarteten Verhalten des System verglichen. Das erwartete Verhalten kann den Testplänen entnommen werden. Gibt es Unterschiede zwischen dem tatsächlichen und dem erwarteten Verhalten des Systems wird dies als Fehlverhalten in der Tabelle zur Bewertung der Testergebnisse im Dokument A-Validation dokumentiert. Anschließend werden die beschriebenen Fehlverhalten auf Fehler in der System-Dokumentation zurückgeführt. Die Fehler werden in der Fehlertabelle im Dokument K-Verifikation festgehalten. Die Tabelle zur Bewertung der Testergebnisse wird um Verweise auf die Fehlernummern erweitert.

Der Kunde entscheidet darüber, ob er das vorliegende installierte System abnimmt, oder nicht. Eventuell akzeptiert der Kunde das System nur unter der Voraussetzung, dass aufgedeckte Fehlverhalten noch im bereits installierten System behoben werden.

Folgende Produkte werden im Prozess *Akzeptanztest durchführen* konsumiert:

- Reale Systemumgebung.
- Installiertes System.

Folgende Produkte werden modifiziert:

- A-Validation.

- K-Verifikation.

Prozess *“Rework Akzeptanztest”*

Im Prozess *Rework Akzeptanztest* werden die in der System-Dokumentation lokalisierten und in der Fehlertabelle im Dokument K-Verifikation dokumentierten Fehler behoben. Die Strategie zur Behebung der Fehler wird in den Verifikationskapiteln der zu ändernden Dokumente beschrieben.

Die Fehler werden entweder im installierten System beim Kunden nachgezogen oder die Korrektur wird mit einem späteren Release des Systems herausgegeben.

Folgende Produkte werden im Prozess Rework Akzeptanztest modifiziert:

- Komponenten-Kode.
- K-Verifikation.
- Ausführbare Komponente.
- Ausführbares System.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

7.7 Prozess “Test im Betrieb”

Im Prozess *Test im Betrieb* wird das beim Kunden installierte System durch die regelmäßige Verwendung getestet. Auftretende Fehlverhalten meldet der Kunde

dem Entwicklungsunternehmen. Dort werden sie im Dokument A-Validation dokumentiert, auf Fehler in der Dokumentation zurückgeführt und behoben. Mit einem späteren Release des Systems können die Korrekturen herausgegeben werden. In eingebetteten Systemen sind gravierende Fehlverhalten, die erst im Prozess “Test im Betrieb” aufgedeckt werden, häufig sehr teuer. Beispielsweise können solche Fehlverhalten mit teuren Rückrufaktionen verbunden sein.

Abbildung 113 zeigt den Prozess *Test im Betrieb* mit konsumierenden, produzierenden und modifizierenden Produkten.

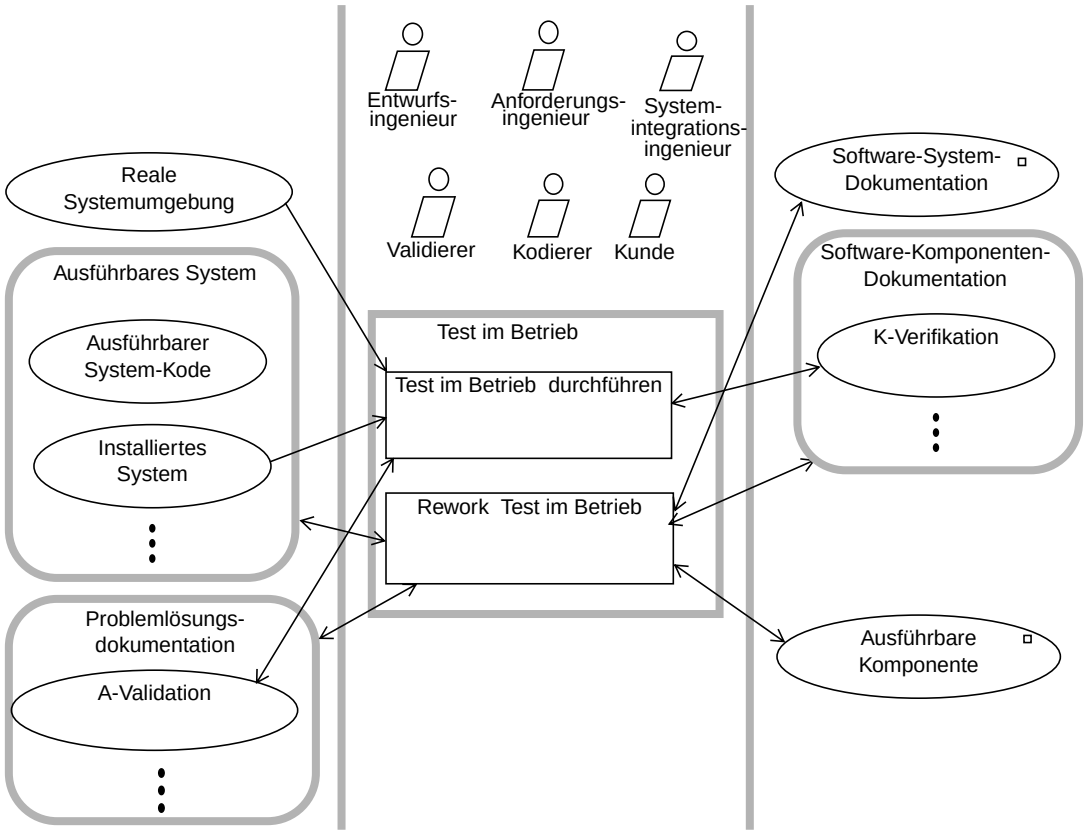


ABBILDUNG 115. Verwendete Produkte des Prozesses *Test im Betrieb*

Prozess *“Test im Betrieb durchführen”*

Im Prozess *Test im Betrieb durchführen* wird das beim Kunden installierte System durch die laufende Verwendung getestet. Der Test im Betrieb ist kein Test im herkömmlichen Sinne. Für die Durchführung werden keine Testbedingungen geschaffen, und es werden keine Tests anhand von Testplänen durchgeführt. Das System wird beim Kunden laufend verwendet und dadurch getestet. Während der Verwendung können Fehlverhalten auftreten, die der Kunde dem Entwicklungsunternehmen meldet.

Das vom System gezeigte und vom Kunden gemeldete Fehlverhalten wird in der “Tabelle zur Bewertung der Testergebnisse” im Dokument A-Validation dokumentiert. Anschließend werden die beschriebenen Fehlverhalten auf Fehler in der System-Dokumentation zurückgeführt. Die Fehler werden in der Fehlertabelle im Dokument K-Verifikation festgehalten. Die Tabelle zur Bewertung der Testergebnisse wird um Verweise auf die Fehlernummern erweitert.

Folgende Produkte werden im Prozess *Test im Betrieb durchführen* konsumiert:

- Reale Systemumgebung.
- Installiertes System.

Folgende Produkte werden modifiziert:

- A-Validation.
- K-Verifikation.

Prozess *“Rework Test im Betrieb”*

Im Prozess *Rework Test im Betrieb* werden die in der System-Dokumentation lokalisierten und in der Fehlertabelle im Dokument K-Verifikation dokumentierten Fehler behoben. Die Strategie zur Behebung der Fehler wird in den Verifikationskapiteln der zu ändernden Dokumente beschrieben.

Die Fehler werden entweder im beim Kunden installierten System nachgezogen, oder die Korrektur wird mit einem späteren Release des Systems herausgegeben.

Folgende Produkte werden im Prozess *Rework Test im Betrieb* modifiziert:

- Komponenten-Kode.
- K-Verifikation.

- Ausführbare Komponente.
- Ausführbares System.
- UML-Entwurf.
- Komponenten-Anforderungen.
- Verfolgbarkeitsmatrix-Entwurf.
- E-Verifikation.
- E-Validation.
- A-Use Cases.
- Szenarien.
- E-Anforderungen.
- Verfolgbarkeitsmatrix-Anforderungen.
- A-Verifikation.
- A-Validation.

7.8 Weiterführende Literatur

1. Peter Liggesmeyer: *Modultest und Modulverifikation: state of the art*. BI-Wiss.-Verlag, Mannheim, Wien, Zürich 1990.
2. Glenford J. Meyers: *Methodisches Testen von Programmen.* Oldenbourg Verlag, München, Wien 1982.

ANHANG A

Produkthierarchie

Im Folgenden werden die Produkte tabellarisch aufgelistet. Die Produkthierarchie wird durch Einrückungen wiedergegeben. Den einzelnen Produkten sind Rollen zugeordnet. Diese Zuordnung spiegelt eine Verantwortlichkeit der Rollen in Bezug auf die Erstellung, Wartung und Bereitstellung der Produkte wider.

Produkt	**Kunde**	**Domänenexperte**	**Anforderungs-ingenieur**	**Entwurfs-ingenieur**	**Verifizierer**	**Validierer**	**Kodierer**	**Systemintegrations-ingenieur**	**Qualitätsmanager**
Problemlösungsdokumentation			X						
Problembeschreibung	X								
B-Anforderungen			X						
Informelle-B-Anforderungen	X		X						
A-Use-Cases			X						
Szenarien			X						
E-Anforderungen			X						
A-Packagediagramme			X						

Produkt	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
A-Klassendiagramme			X						
Data-Dictionary			X						
A-Sequenzdiagramme			X						
A-Kollaborationsdiagramme			X						
A-Zustandsdiagramme			X						
A-Aktivitätsdiagramme			X						
Verfolgbarkeitsmatrix-Anforderungen			X						
A-Verifikation					X				
A-Validation						X			
Software-System-Dokumentation				X					
Entwurfsentscheidungen				X					
UML-Entwurf				X					
Unterschiede zur Analyse				X					
E-Klassendiagramme				X					
Class-Tables				X					
E-Use-Cases				X					
E-Sequenzdiagramme				X					
E-Kollaborationsdiagramme				X					
E-Zustandsdiagramme				X					
E-Aktivitätsdiagramme				X					
E-Packagediagramme				X					
Verfolgbarkeitsmatrix-Entwurf				X					
Komponenten-Anforderungen				X					
E-Verifikation					X				
E-Validation						X			
Software-Komponenten-Dokumentation							X		
Komponenten-Kode							X		
K-Verifikation					X				

Produkt	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
Ausführbare-Komponente							X		
Komponententest-Rahmen						X			
Komponententest-Stubs						X			
Komponententest-Treiber						X			
Ausführbarer Komponenten-Kode							X		
Ausführbares-System								X	
Ausführbares Teilsystem								X	
Teilsystem-Makefile								X	
Ausführbarer Teilsystem-Kode								X	
Teilsystemtest-Rahmen						X			
Ausführbarer System-Kode								X	
Installiertes System								X	
Entwicklungsrichtlinien									X
Kode-Ableitungsrichtlinien									X
Programmierrichtlinien									X
Domänenwissen		X							
Systemumgebung		X							
Referenz-Dokumente		X							
Namenskonventionen									X
Simulator						X			
Reale Systemumgebung	X								

ANHANG B

Prozesshierarchie

Im Folgenden werden die Prozesse tabellarisch aufgelistet. Zu den einzelnen Prozessen werden die beteiligten Rollen angegeben.

Prozess	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
Projekt initialiseren	X	X	X						
Problembeschreibung bearbeiten	X								
Informelle B-Anforderungen bearbeiten	X		X						
Domänenwissen bearbeiten		X							
System-Anforderungen bearbeiten	X	X	X		X	X			X
B-Anforderungen bearbeiten	X		X						
A-Use-Cases bearbeiten	X		X						
Szenarien bearbeiten	X		X						
E-Anforderungen bearbeiten			X						
A-Sequenzdiagramme bearbeiten			X						

Prozess	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
A-Klassendiagramme bearbeiten			X						
A-Zustandsdiagramme bearbeiten			X						
A-Aktivitätsdiagramme bearbeiten			X						
A-Kollaborationsdiagramme bearbeiten			X						
Data-Dictionary bearbeiten			X						
Verfolgbarkeitsmatrix-Anforderungen bearbeiten			X						
System-Anforderungen verifizieren	X		X		X				
System-Anforderungen Review vorbereiten					X				
Reviewsitzung System-Anforderungen	X		X		X				
Rework System-Anforderungen			X						
Testfälle bearbeiten						X			
System-Entwurf bearbeiten			X	X	X	X			X
Entwurfentscheidungen erstellen				X					
UML-Entwurf bearbeiten			X	X					
E-Use-Cases bearbeiten			X	X					
E-Klassendiagramme bearbeiten			X	X					
Class-Tables-bearbeiten			X	X					
E-Zustandsdiagramme bearbeiten			X	X					
E-Aktivitätsdiagramme bearbeiten			X	X					
E-Sequenzdiagramme bearbeiten			X	X					
E-Kollaborationsdiagramme bearbeiten			X	X					
Komponenten-Anforderungen erstellen				X					
Verfolgbarkeitsmatrix-Entwurf bearbeiten				X					
System-Entwurf verifizieren			X	X	X				
System-Entwurf Review vorbereiten					X				
Reviewsitzung System-Entwurf				X	X				
Rework System-Entwurf			X	X					
Komponenten-Testfälle bearbeiten						X			

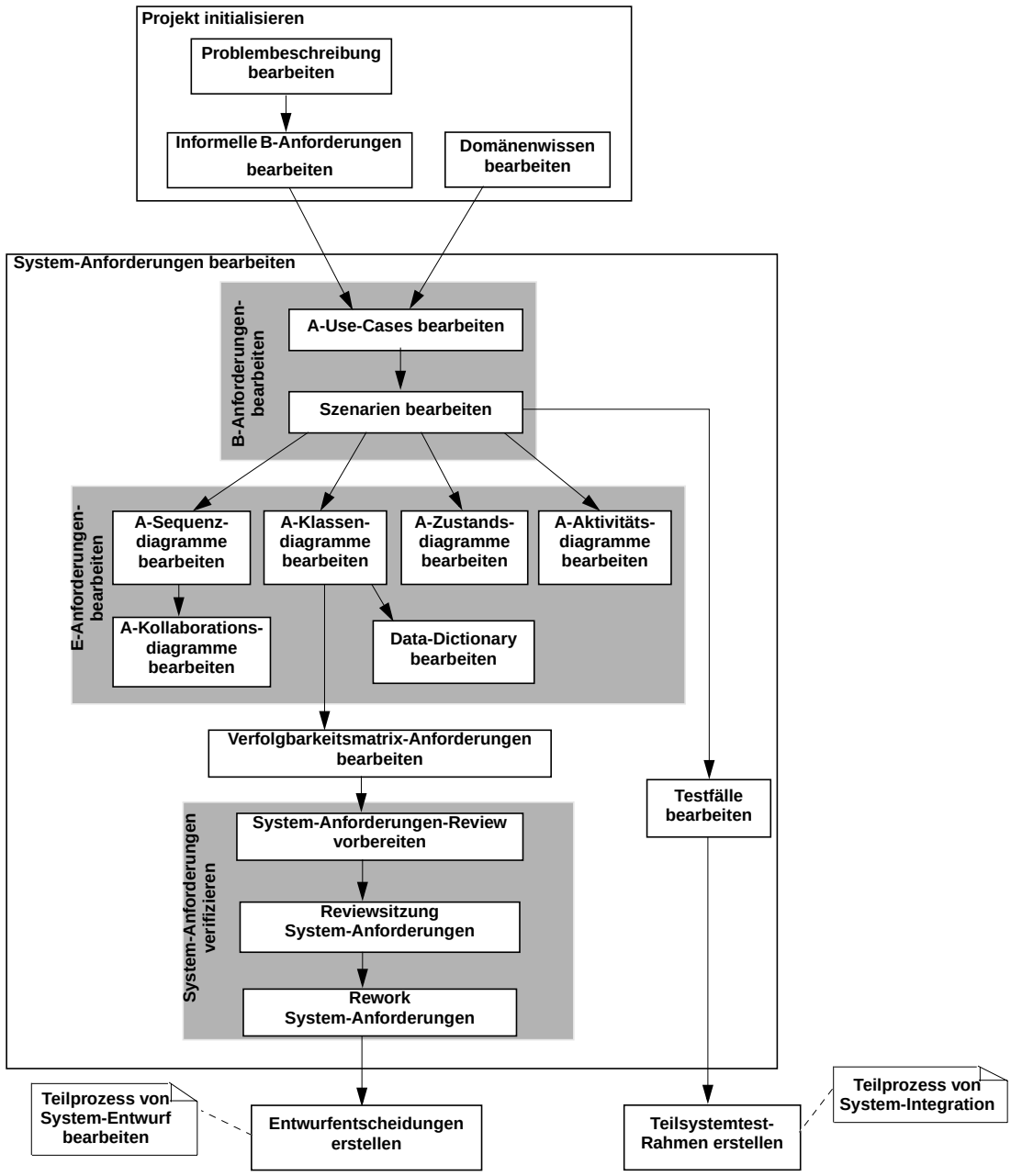
Prozess	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
System-Komponenten bearbeiten			X	X	X	X	X		X
Quellkode bearbeiten			X	X	X		X		
Komponenten-Kode bearbeiten							X		
Komponenten-Kode verifizieren			X	X	X		X		
Komponenten-Kode Review vorbereiten					X				
Reviewsitzung Komponenten-Kode					X		X		
Rework Komponenten-Kode			X	X			X		
Ausführbare Komponente bearbeiten			X	X		X	X		
Komponententest-Rahmen erstellen						X			
Komponententest-Stubs erstellen						X			
Komponententest-Treiber erstellen						X			
Ausführbare Komponente erstellen							X		
Komponententest durchführen						X			
Rework Komponententest			X	X			X		
System-Integration			X	X		X	X	X	X
Integriere Teilsystem						X		X	
Teilsystemtest-Rahmen erstellen						X			
Teilsystem-Makefile erstellen								X	
Ausführbares Teilsystem erstellen								X	
Integrationstest			X	X		X	X		
Teilsystemtest durchführen						X			
Rework Teilsystem			X	X			X	X	
System-Validation	X		X	X		X	X	X	X
Systemtest			X	X		X	X	X	
Systemtest durchführen						X			
Rework Systemtest			X	X			X	X	
Akzeptanztest	X		X	X		X	X	X	
Installiere System	X							X	

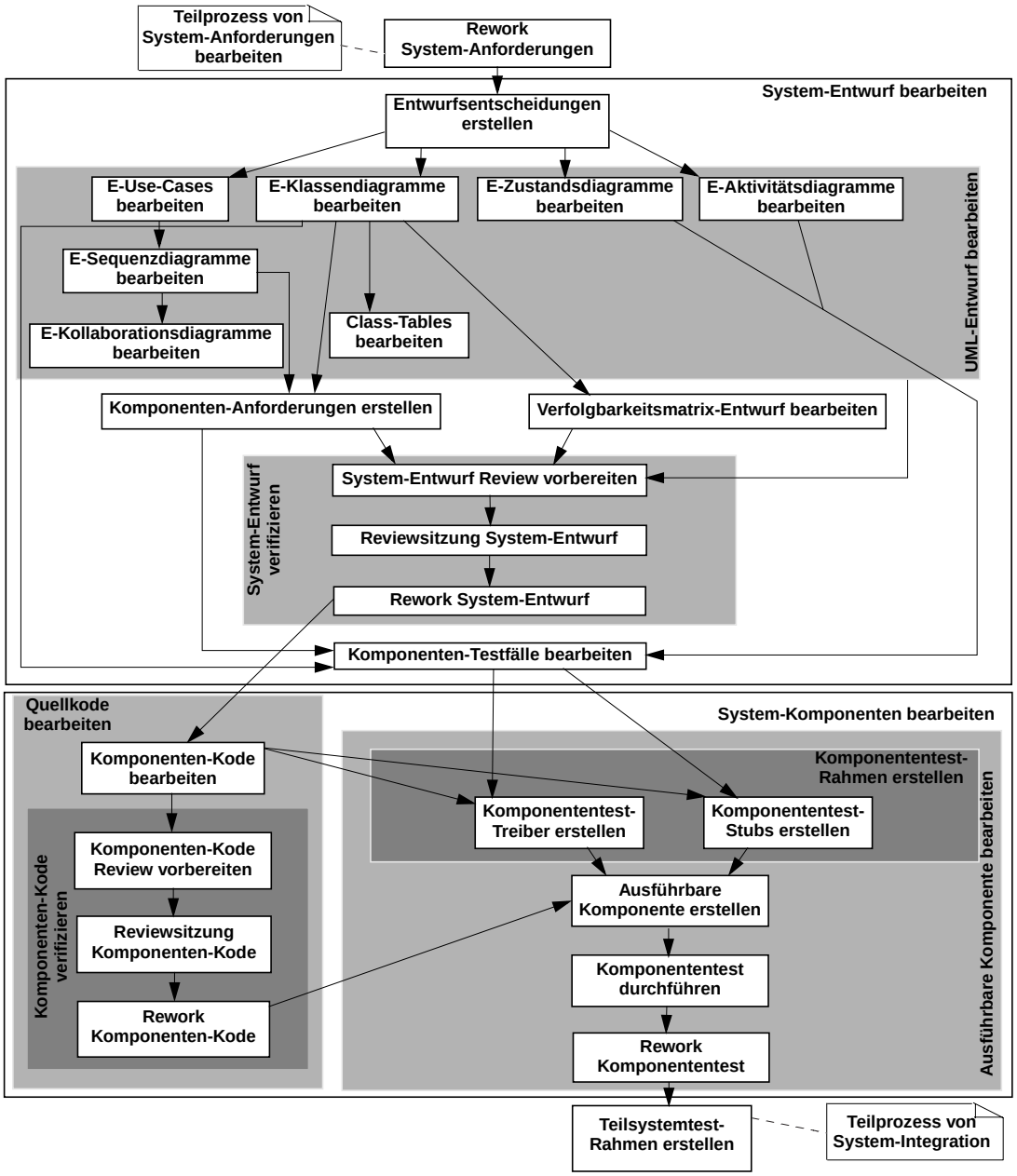
Prozess	Kunde	Domänenexperte	Anforderungs-ingenieur	Entwurfs-ingenieur	Verifizierer	Validierer	Kodierer	Systemintegrations-ingenieur	Qualitätsmanager
Akzeptanztest durchführen	X					X			
Rework Akzeptanztest			X	X			X	X	
Test im Betrieb	X		X	X		X	X	X	
Test im Betrieb durchführen	X					X			
Rework Test im Betrieb			X	X			X	X	

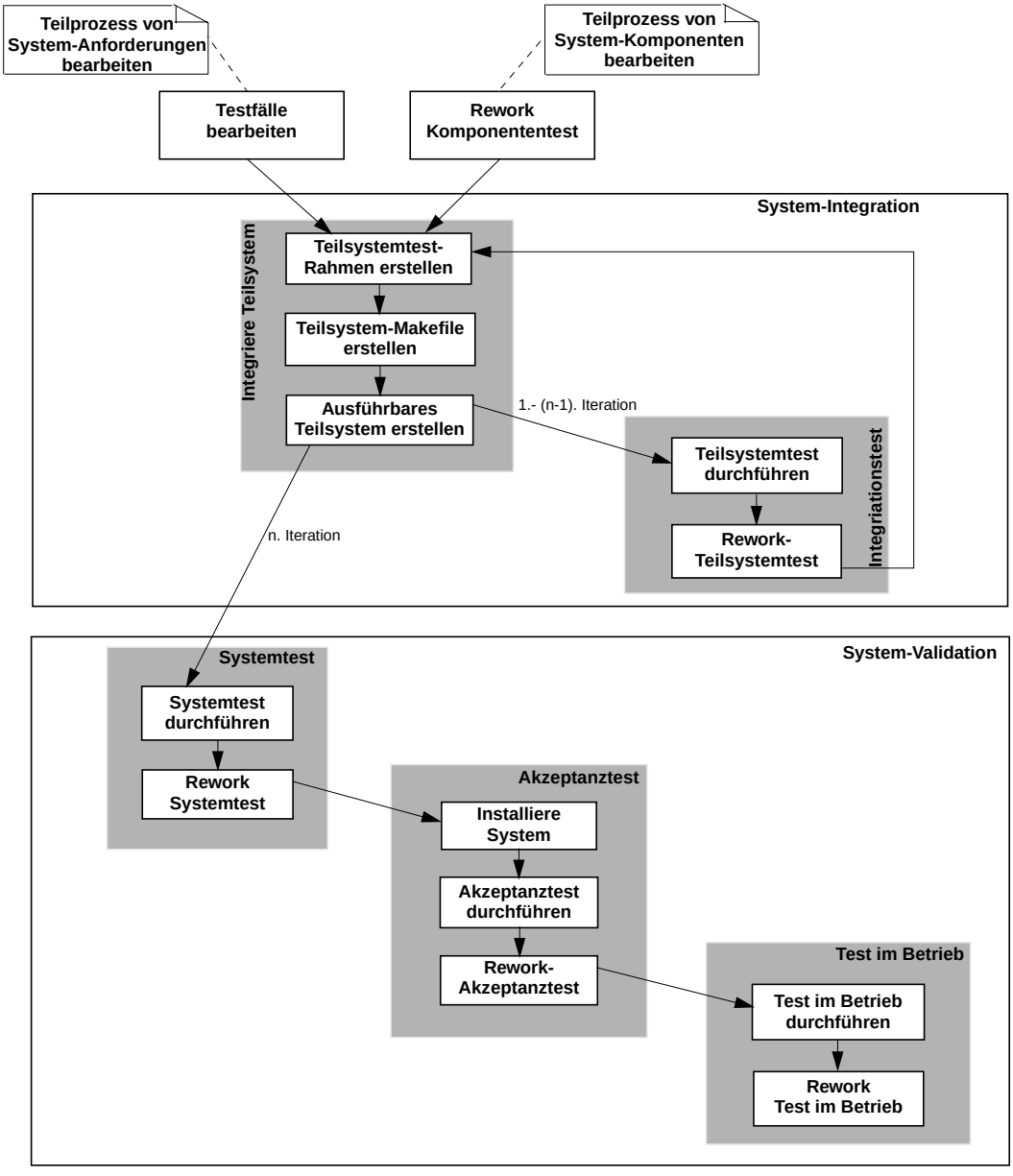
ANHANG C

Kontrollfluss

Im Folgenden wird die Abarbeitungsreihenfolge der einzelnen Prozesse in Form eines Präzedenzgraphen angegeben. Nähere Angaben hierzu können den entsprechenden Kapiteln entnommen werden.


Projekt initialisieren
Problembeschreibung bearbeiten
Informelle B-Anforderungen bearbeiten
Domänenwissen bearbeiten
System-Anforderungen bearbeiten
B-Anforderungen-bearbeiten
A-Use-Cases bearbeiten
Szenarien bearbeiten
E-Anforderungen-bearbeiten
A-Sequenz-diagramme bearbeiten
A-Klassen-diagramme bearbeiten
A-Zustands-diagramme bearbeiten
A-Aktivitäts-diagramme bearbeiten
A-Kollaborations-diagramme bearbeiten
Data-Dictionary bearbeiten
Verfolgbarkeitsmatrix-Anforderungen bearbeiten
Testfälle bearbeiten
System-Anforderungen verifizieren
System-Anforderungen-Review vorbereiten
Reviewsitzung System-Anforderungen
Rework System-Anforderungen
Teilprozess von System-Entwurf bearbeiten
Entwurfentscheidungen erstellen
Teilsystemtest-Rahmen erstellen
Teilprozess von System-Integration


Teilprozess von System-Anforderungen bearbeiten
Rework System-Anforderungen
System-Entwurf bearbeiten
Entwurfsentscheidungen erstellen
E-Use-Cases bearbeiten
E-Klassendiagramme bearbeiten
E-Zustandsdiagramme bearbeiten
E-Aktivitätsdiagramme bearbeiten
UML-Entwurf bearbeiten
E-Sequenzdiagramme bearbeiten
E-Kollaborationsdiagramme bearbeiten
Class-Tables bearbeiten
Komponenten-Anforderungen erstellen
Verfolgbarkeitsmatrix-Entwurf bearbeiten
System-Entwurf verifizieren
System-Entwurf Review vorbereiten
Reviewsitzung System-Entwurf
Rework System-Entwurf
Komponenten-Testfälle bearbeiten
Quellkode bearbeiten
System-Komponenten bearbeiten
Komponenten-Kode bearbeiten
Komponententest-Rahmen erstellen
Komponententest-Treiber erstellen
Komponententest-Stubs erstellen
Komponenten-Kode verifizieren
Komponenten-Kode Review vorbereiten
Reviewsitzung Komponenten-Kode
Rework Komponenten-Kode
Ausführbare Komponente erstellen
Komponententest durchführen
Rework Komponententest
Ausführbare Komponente bearbeiten
Teilsystemtest-Rahmen erstellen
Teilprozess von System-Integration


Teilprozess von System-Anforderungen bearbeiten
Teilprozess von System-Komponenten bearbeiten
Testfälle bearbeiten
Rework Komponententest
System-Integration
Integriere Teilsystem
Teilsystemtest-Rahmen erstellen
Teilsystem-Makefile erstellen
Ausführbares Teilsystem erstellen
1.- (n-1). Iteration
Integriationstest
Teilsystemtest durchführen
Rework-Teilsystemtest
n. Iteration
System-Validation
Systemtest
Systemtest durchführen
Rework Systemtest
Akzeptanztest
Installiere System
Akzeptanztest durchführen
Rework-Akzeptanztest
Test im Betrieb
Test im Betrieb durchführen
Rework Test im Betrieb

Index

A

A-Aktivitätsdiagramme 56
A-Aktivitätsdiagramme bearbeiten 80
A-Checkliste 59
A-Ergebnisbewertung 66
A-Fehlerbehebungsstrategien 61
A-Fehlertabellen 60
A-Klassendiagramme 48
A-Klassendiagramme bearbeiten 76
A-Kollaborationsdiagramme 53
A-Kollaborationsdiagramme bearbeiten 79
Akzeptanztest 220
Akzeptanztest durchführen 222
Anforderungsingenieur 31, 68, 127, 169, 199, 215
A-Packagediagramme 46
A-Sequenzdiagramme 51
A-Sequenzdiagramme bearbeiten 78
A-Testergebnisse 65
A-Testpläne 64
A-Testverfahren 63
A-Use Cases 42
A-Use Cases bearbeiten 72
Ausführbare Komponente 152, 166, 188, 210
Ausführbare Komponente bearbeiten 179
Ausführbarer Komponenten-Kode 169
Ausführbarer System-Kode 198
Ausführbarer Teilsystem-Kode 198
Ausführbares System 187, 192, 210
Ausführbares Teilsystem 193
Ausführbares Teilsystem erstellen 204
A-Validation 62
A-Verifikation 58
A-Zustandsdiagramme 54
A-Zustandsdiagramme bearbeiten 79

B

B-Anforderungen 26, 41
B-Anforderungen bearbeiten 70
Baseline 2

C
Class-Tables 103
Class-Tables bearbeiten 135

D
Data-Dictionary 50
Data-Dictionary bearbeiten 78
Domänenexperte 30, 69
Domänenwissen 28
Domänenwissen bearbeiten 32

E
E-Aktivitätsdiagramme 108
E-Aktivitätsdiagramme bearbeiten 138
E-Anforderungen 46
E-Anforderungen bearbeiten 74
E-Checkliste 115
E-Ergebnisbewertung 125
E-Fehlerbehebungsstrategien 118
E-Fehlertabellen 116
E-Klassendiagramme 101
E-Klassendiagramme bearbeiten 134
E-Kollaborationsdiagramme 106
E-Kollaborationsdiagramme bearbeiten 137
Entwicklungsrichtlinien 162
Entwurfsentscheidungen 96
Entwurfsentscheidungen erstellen 129
Entwurfsingenieur 169, 199, 215
E-Packagediagramme 99
E-Sequenzdiagramme 105
E-Sequenzdiagramme bearbeiten 136
E-Testergebnisse 123
E-Testpläne 120
E-Testverfahren 119
E-Use Cases 104
E-Use Cases bearbeiten 134
E-Validation 119
E-Verifikation 114
E-Zustandsdiagramme 107
E-Zustandsdiagramme bearbeiten 137

F
Fehlerklassifikation 59, 116, 159

I
Informelle B-Anforderungen 26
Informelle B-Anforderungen bearbeiten 34
Installiere System 221
Installiertes System 198
Integrationstest 204
Integriere Teilsystem 201

K
K-Checkliste 159
K-Fehlerbehebungsstrategien 161
K-Fehlertabellen 160
Kode-Ableitungsrichtlinien 162
Kodierer 169, 199, 215
Komponente 91, 92, 93
Komponenten-Anforderungen 111
Komponenten-Anforderungen erstellen 139
Komponenten-Kode bearbeiten 174
Komponenten-Kode Review vorbereiten 177
Komponenten-Kode verifizieren 174
Komponenten-Testfälle bearbeiten 146
Komponententest-Rahmen 166
Komponententest-Rahmen erstellen 181
Komponententest-Stubs 167
Komponententest-Treiber 166
Kunde 30, 68, 215
K-Verifikation 158

N
Namenskonventionen 67, 126

P
Package 99
Problembeschreibung 25
Problembeschreibung bearbeiten 32
Problemlösungsdokumentation 38, 92, 152, 187, 210
Produkt 8
Programmierrichtlinien 164
Projekt initialisieren 21

Prozess 10

Q
Qualitätsmanager 69, 127, 169, 199, 215
Quellkode bearbeiten 171

R
Reale Systemumgebung 210, 214
Referenz-Dokumente 29
Reviewsitzung Komponenten-Kode 177
Reviewsitzung System-Anforderungen 86
Reviewsitzung System-Entwurf 145
Rework Akzeptanztest 223
Rework Komponenten-Kode 178
Rework System-Anforderungen 86
Rework System-Entwurf 145
Rework Systemtest 219
Rework Teilsystem 206
Rework Test im Betrieb 225
Rolle 12

S
Simulator 209
Software-Komponenten-Dokumentation 152, 188, 210
Software-System-Dokumentation 92, 152, 188, 210
System-Anforderungen bearbeiten 37
System-Anforderungen Review vorbereiten 85
System-Anforderungen verifizieren 83
System-Entwurf bearbeiten 91
System-Entwurf Review vorbereiten 144
System-Entwurf verifizieren 142
System-Integration 187
Systemintegrationsingenieur 199, 215
System-Komponenten bearbeiten 151
Systemtest 217
Systemtest durchführen 219
Systemumgebung 28
System-Validation 209
Szenarien 45
Szenarien bearbeiten 73

T
Teilsystem 99
Teilsystem-Makefile 197
Teilsystem-Makefile erstellen 203
Teilsystemtest durchführen 206
Teilsystemtest-Rahmen 193
Teilsystemtest-Rahmen erstellen 203
Test im Betrieb 223
Test im Betrieb durchführen 225
Testfälle bearbeiten 87

U
UML-Entwurf 98
UML-Entwurf bearbeiten 130
Unterschiede zur Analyse 98

V
Validation 3
Validierer 69, 127, 169, 199, 215
Verfolgbarkeitsmatrix-Anforderungen 57
Verfolgbarkeitsmatrix-Anforderungen bearbeiten 81
Verfolgbarkeitsmatrix-Entwurf 110
Verfolgbarkeitsmatrix-Entwurf bearbeiten 141
Verifikation 3
Verifizierer 69, 126, 169